Anonymous (co)inductive types: A way for structured recursion to cohabit with modular abstraction

Mikołaj Konarski
Institute of Informatics, Warsaw University
Banacha 2, 02-097 Warszawa, Poland
mikon@mimuw.edu.pl

Abstract

We investigate the interaction between structured recursion combinators and modularization in the style of Standard ML. When built-in structured recursion combinators are straightforwardly added to a language like SML'97 or OCaml, they cannot operate over values of abstractly specified types. Consequently, when a program is modularized in an abstract and fine-grained way, the structured recursion combinators can hardly ever be used. We explore various ways of solving this incompatibility, presenting the possible solutions in our modular programming language with a verbose notation for inductive and coinductive types and operations. We propose structured recursion programming constructs that are, in our opinion, best suited for action across abstract module boundaries: anonymous inductive and coinductive types separated from sum and product types, and the related operations. We discuss advantages and disadvantages of the proposed programming idioms and their expressibility in other languages and formal systems, such as OCaml, Charity, Functorial ML and PolyP. We describe their precise typing and semantics as parts of our programming language. Finally we discuss other possible uses of the introduced constructs, related to both polytypism and modular programming and inspired by the notions of views and 2-level types.

Keywords: structured recursion, module systems, abstraction, category theory, (co)inductive types

1. STRUCTURED RECURSION ACROSS MODULES

The problem of structured recursion over abstract types is not difficult to solve, but in our opinion it is interesting, important and the various ways of solving it are, as far as we know, unexplored. In this section we argue about advantages and disadvantages of different solutions through examples. We present our examples using the syntax of programming language Dule [9]. The language has a light-weight module system and a verbose notation for the inductive and coinductive operations. This allows us to easily present many variants of construction and specification of (co)inductive values.

The syntax of the core language is close to that of OCaml [10], including its recent notational novelties. In particular, function parameters are named by labels adorned with tildes and anonymous sum type variant names begin with back-quotes. Neither the labels nor the variant names have scopes or types assigned within a scope. All the types — function types, sum types, record types, (co)inductive types — are anonymous, that is, they have no identity other than their construction. The labels and variant names have no semantics outside types or their related operations, and they can appear in many types at once. Type reconstruction resolves emerging ambiguities.

We always mark labels of function and module parameters with tilde, as in `~n` and retain the tilded labels in typing (in OCaml the tildes are usually optional). However, in function bodies the parameters are referred to without `~`, as in `n`. The phrase `Nat.leq ~n:n ~it:1`, abbreviated to `Nat.leq ~n ~it:1`, is an application of a function `leq` from the module `Nat` to the arguments `n`

and `1` at positions labeled `n` and `it`, respectively. Label `it` appears in the typing of many built-in operations, but the label has no special status and can be freely used by the programmer — in fact, there are no reserved labels nor variant names in Dule. Names of variants of anonymous sum types are marked, similarly as in OCaml, by back-quote and are always written uppercase, as in `‘True`. The sum types themselves are written between square brackets, just as is case analysis. A brief description of the semantics of the core language of Dule and the typing rules for all structured recursion constructs used below are given in Section 2. A full language definition, with a tutorial and theoretical analysis of the semantics can be found in [9].

Every example given in this paper can be successfully type-checked and compiled using the Dule compiler [8]. The file gathering all the examples is `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/test/anonymous.dul` and can be processed by issuing "`./dule ../test/anonymous.dul`" or "`make test-anonymous`".

1.1. Difficulties

1.1.1. Structured recursion within a module

Let us consider a specification (parameterized signature) and a module of lists with elements of type `Elem.t`. Module `List` is implicitly typed with the signature of the same name, so the module has parameter `Elem`.

```
spec List =
~Elem:sig type t end ->
  sig
    type t
    value nil : t
    value cons : ~head:Elem.t ~tail:t -> t
  end
module List =
  struct
    type t = ind list: [‘Nil|‘Cons {head : Elem.t; tail : list}]
    value nil = ‘Nil . con
    value cons = fun ~head ~tail -> {head = head; tail = tail} .‘Cons . con
  end
```

Type `t` is defined above as inductive closure of a sum type with variants `‘Nil` and `‘Cons`. Value `nil` is defined as a constructor of the inductive type corresponding to variant `‘Nil` of the sum type. Value `cons` is a function that produces a record of its arguments, taken into the sum type by composing with injection `‘Cons` and then taken into the inductive type of lists by composing with the inductive type constructor combinator `con`.

Inside the module we can use induction (`fold`) over the inductive type. For example we can add a list catenation function to the signature and to the module implementation, as follows. (In case of problems with the new syntax, compare with the implementation of `append` written in a more conventional style in the next section.)

```
    value append : ~l1:t ~l2:t -> t

    value append = fun ~l1 ~l2 ->
      match l1 with
      fold [‘Nil -> l2
           |‘Cons ht -> ht .‘Cons . con]
```

However, `append` couldn't be defined outside module `List`, because type `t` in the signature is specified as abstract and the operations provided in the signature enable only construction of lists, not their destruction; in particular no case analysis is possible.

If modules are truly abstract, as is the case in Dule, OCaml, SML'97 [12] (as opposed to SML'90 [11]), then (co)inductive combinators cannot act on abstractly specified types. To enable structured recursion we could make the type of lists transparent using a type abbreviation. However, then all benefits of abstraction are lost — in particular, implementations of the list type even slightly different than the specified implementation can no longer be made to fit the signature (see the type of lists and their length in the example in Section 1.2.2). Consequently, a programmer using the module with transparent type cannot abstract from the details of the type's implementation. A better solution would be to provide two specifications of types of lists: one abstract, the other transparent. We will present a similar, but cleaner approach using anonymous inductive types in Section 1.2.1.

1.1.2. Deconstruction of an inductive type

To maintain abstraction but enable deconstruction of lists, we can add to the module of lists a conversion operation. The operation `tde` converts a value of the abstract list type to a value of a corresponding sum type, where the `tail` component is again the list type. Notice that the result type of `tde` is not the same as the inductive type of lists! Operations `tde` destructs a list by composing it with combinator `de`, which is the categorical isomorphism inverse to `con`.

```
    value tde : ~it:t -> [‘Nil|‘Cons {head : Elem.t; tail : t}]

    value tde = fun ~it -> it . de
```

Conversion `tde` allows us to use case expression and recursive definitions by case analysis (`value rec` below). However, this is not an instance of structured recursion, even if the pattern of recursion is the same as in the definition using `fold` above, because the recursive call to `append` is here performed explicitly.

```
spec ListOps =
~List ->
  sig
    value append : ~l1:List.t ~l2:List.t -> List.t
  end
module ListOps =
  struct
    value rec append = fun ~l1 ~l2 ->
      match List.tde ~it:l1 with
      [‘Nil -> l2
      |‘Cons {head; tail} ->
          List.cons ~head ~tail:(append ~l1:tail ~l2)] (* recursive call! *)
  end
```

In cases when we want to operate on values of a particular inductive type only with general recursion, `tde` as above is all we need. An abstract type with only constructors and `tde` is guaranteed to be used in a program only in a conventional functional way — without structured recursion. "Polymorphic variants" of OCaml can be used to easily specify and define functions similar to `tde`. Strictly speaking, the OCaml variant types are inductive types merged with sum types, but since OCaml does not have structured recursion combinators, the behavior of the OCaml `tde` and Dule `tde` would be analogous. In Charity [3] the inductive and sum types are merged, as in conventional functional programming languages, but the language has no general recursion, so a distinction between the type constructors would be mostly pointless.

In PolyP [4], as in underlying Haskell [13], there is no way to express sum types outside of inductive (recursive, in fact) type closure. On the other hand, PolyP has both structured and general recursion mechanisms, so the ability to mark abstract types for one or the order kind of recursion would be useful. The PolyP counterpart of `tde` (or rather of our built-in destructor combinator `de`), called `out`, is typed using an additional sum-recursive type with void type closure.

The unreadable types of instances of `out` are not problematic for polytypic programming inside modules, but they do not seem acceptable for specifications of modules.

Functorial ML [6] offers a finer distinction, similar to the one in Dule: there are distinctive sum and inductive type constructors and term constructor `unwrap` analogous to `tde`. However, Functorial ML is described as only an intermediate language, and indeed `unwrap` seems to vanish in examples of Functorial ML programs written in ML-like notation. We would suggest that the distinction between sum and inductive types is not forgone when Functorial ML is equipped with syntactic sugar or integrated into Standard ML. Perhaps the Dule notation may suggest ways to keep the distinction without overburdening the programmer.

1.1.3. Manually defined combinators

We would like to retain abstraction of the list type, at the same time enabling structured recursion over lists. This can be done by defining an iterator as a value in module `List`. In the following code, for simplicity, we present only a monomorphic version of the iterator `foldf`.

```
value foldf : ~f:~e:Elem.t ~acc:t -> t
              ~init:t
              ~l:t -> t

value foldf = fun ~f ~init ~l ->
  match l with
  fold [‘Nil -> init
       |‘Cons {head; tail} -> f ~e:head ~acc:tail]
```

However, definitions by structured recursion using `foldf` are cumbersome, because one cannot use the notation for case analysis. If the implementation of the abstract type contains a sum with many variants, as is the case, e.g., with types representing grammars, the iterator function receives a multitude of parameters making it especially unreadable.

We can do better by using anonymous sum types (closed sum type expressions). Again we present only the monomorphic case.

```
value foldr : ~f:~it:[‘Nil|‘Cons {head : Elem.t; tail : t}] -> t
              ~l:t -> Elem.t

value foldr = fun ~f ~l ->
  match l with
  fold f
```

With the help of function `foldr` we can define functions by structured recursion over abstract types using the convenient notation for case analysis, as in the following example.

```
spec ListOps =
~List ->
  sig
    value append : ~l1:List.t ~l2:List.t -> List.t
  end
module ListOps =
  struct
    value append = fun ~l1 ~l2 ->
      List.foldr
        ~f:[‘Nil -> l2
           |‘Cons {head; tail} -> List.cons ~head ~tail]
        ~l:l1
  end
```

Actually, OCaml mixed sum/inductive types can be used for `foldr` as well. This time, though, the semantic mix-up of sum and inductive types is quite glaring. The domain of the functional argument to a structured recursion combinator is supposed to be a sum type, not an inductive type as in OCaml. In particular, the user is not supposed to define the functional argument to an iterator by recursion over the inductive type, which is possible when using OCaml "polymorphic variants" to specify the type of `foldr`.

For full generality, the `foldr` approach requires polymorphism not only in the core language but also in module signatures. Moreover, for each kind of a polytypic structured recursion combinator of the programming language at hand (such as `fold`, `map`, `zip`, etc.) we would need corresponding signature and module entries. If we intend to also use general recursion over our abstract type, we additionally need `tde` as well. This can even be seen as an advantage of the `foldr` approach, since the verbosity gives us some control on the kind of recursion allowed over the abstract type of the defined module.

However, the real problem is that the polytypic flavor [7] of structured recursion is utterly lost. The `foldr` and similar functions have to be defined anew and with different typing for each different inductive type in a program. As before, neither built-in nor user defined (if a language supports them) polytypic combinators can act over the values constructed with `List` module's operations.

1.2. Proposed approach

1.2.1. Anonymous inductive types

We propose an approach to structured recursion across modules that (in the simplest case) assures full power of polytypic combinators, as if there was no module boundaries. At the same time our approach, based on the ability to write inductive types (`ind...`) anonymously in signatures, does not violate the abstraction of the types defined in modules and allows their different but compatible implementation to look the same to the outside world. Here is our advocated version of the signature and module `List`. In the last line of the module's implementation we have written the record `{head = head; tail = tail}` in an abbreviated form, as `{head; tail}`.

```
spec List =
~Elem:sig type t end ->
  sig
    type t
    value t2ind : ~it:t -> ind list: [‘Nil|‘Cons {head : Elem.t; tail : list}]
    value nil : t
    value cons : ~head:Elem.t ~tail:t -> t
  end
module List =
  struct
    type t = ind list: [‘Nil|‘Cons {head : Elem.t; tail : list}]
    value t2ind = fun ~it -> it (* identity! *)
    value nil = ‘Nil . con
    value cons = fun ~head ~tail -> {head; tail} .‘Cons . con
  end
```

In the simplest case, the type conversion operation `t2ind` is implemented as the identity function, despite the complex typing of its codomain. See next section for a case of `t2ind` that is not an identity, but has the same typing as above.

Lists equipped with the conversion to the anonymous inductive type can be used as follows:

```
spec ListOps =
~List ->
  sig
```

```
    value append : ~l1:List.t ~l2:List.t -> List.t
    value is_nil : ~l:List.t -> [‘True|‘False]
  end
module ListOps =
  struct
    value append = fun ~l1 ~l2 ->
      match List.t2ind ~it:l1 with
      fold [‘Nil -> l2
           |‘Cons {head; tail} -> List.cons ~head ~tail]
    value is_nil = fun ~l ->
      match (List.t2ind ~it:l) . de with
      [‘Nil -> ‘True
      |‘Cons -> ‘False]
  end
```

In function `is_nil` we see the expression `(List.t2ind ~it:l) . de`, which is not equivalent to `tde ~it:l` from the previous sections, but for this application it suffices. Operation `tde` can be expressed using `t2ind`, `nil` and `cons`, but in a complicated way (see Section 1.4). For performing general recursion in the absence of `tde`, usually a less expensive solution is to operate solely on the result of `t2ind`, without references to the abstract type. In this way, `t2ind` enables both structured recursion and general recursion, although the latter is typed in a less clean way.

If we plan on using values of the abstract inductive type mainly with general recursion, or on providing complex implementations, where `t2ind` is expensive and `tde` is not, we can equip the signature with both `t2ind` and `tde`. The separation of inductive and sum types in the language ensures that the typing of the two conversion operations will be distinct, according to their functionality, and that the their code can be accordingly optimized. In different parts of the program the module can be seen with a smaller specification, depending on whether it is to be used locally with structured or general recursion.

In OCaml, in the absence of structured recursion combinators, `t2ind` is equivalent to `tde`. Charity does not have anonymous datatypes and its notation for datatypes, though suggestive and readable, results in quite long definitions. However, by introducing local datatype definitions in signatures, we could provide a typing for `t2ind`. We are not aware of any extension of Haskell with anonymous datatypes or “polymorphic variants”, so PolyP types cannot be used for `t2ind`. PolyP functors are anonymous, but even more so than desired — they do not record variant names for sum types, for instance. They are not designed to type values, but to define polytypic operators.

In Functorial ML functors are anonymous, as in PolyP, but types are defined using functors, so there is enough notation (even if quite low-level) to implement values of the types constructed with functors. What is disturbing, however, is the intentional interpretation of functors. For example, functor composition is not associative, though suitable isomorphisms are provided as values of the language. The intentionality is useful for directing type reconstruction of the mapping combinator, so that it becomes unambiguous (compare with Fact 2.1), but intentional specification of modules creates paradoxes (e.g., you have to write isomorphisms to compare a list of (pairs of integers) with a (list of pairs) of integers). However, it is possible that the intentionality is meant to vanish in the final semantics — Functorial ML is said to be an intermediate language. Then matching modules to specifications is again simple and intuitive, but separately compiled modules with inductive types and `t2ind` conversions loose their ability to direct type reconstruction of mapping.

1.2.2. Abstraction with anonymous inductive types

To see how our approach adapts to type implementation changes, let us consider an extension of the module specification `List` with a parameter `Nat` (a module of natural numbers with basic operations) and operation `length` computing the length of a list.

```
spec LengthList =
~Nat ~Elem:sig type t end ->
  sig
    type t
    value t2ind : ~it:t -> ind list: [‘Nil|‘Cons {head : Elem.t; tail : list}]
    value nil : t
    value cons : ~head:Elem.t ~tail:t -> t
    value length : ~l:t -> Nat.t
  end
```

The module can be implemented as an extension of our module of ordinary lists.

```
module LengthList =
  struct
    type t = ind list: [‘Nil|‘Cons {head : Elem.t; tail : list}]
    value t2ind = fun ~it -> it
    value nil = ‘Nil . con
    value cons = fun ~head ~tail -> {head; tail} .‘Cons . con
    value length = fun ~l ->
      match l with
      fold [‘Nil -> Nat.zero
           |‘Cons {head; tail} -> Nat.succ ~n:tail]
  end
```

We can also provide a more sophisticated implementation, based on a different type of lists. Notice that the module fits the same signature `LengthList` and its behavior (in particular with respect to polytypic structured recursion combinators) is indistinguishable from the behavior of the simpler implementation, apart of the better asymptotic complexity of `length`.

```
module LengthList =
  struct
    type t = {n : Nat.t;
              l : ind list: [‘Nil|‘Cons {head : Elem.t; tail : list}]}
    value t2ind = fun ~it:{n; l} -> l (* not identity! *)
    value nil = {n = Nat.zero;
                 l = ‘Nil . con}
    value cons = fun ~head ~tail:{n; l} ->
      {n = Nat.succ ~n;
       l = {head; tail = l} .‘Cons . con}
    value length = fun ~l:{n; l} -> n
  end
```

Conversions `t2ind` belonging to different datatypes can be composed as in the following, third and the last, implementation of module `LengthList`. We see here an explicit typing of the module, adding a parameter module `List` (matching the specification `List` defined in the previous section).

```
module LengthList =
  :: ~List -> LengthList
  struct
    type t = {n : Nat.t; l : List.t}
    value t2ind =
      let local_t2ind = fun ~it:{n; l} -> l in
      fun ~it -> List.t2ind ~it:(local_t2ind ~it) (* composition *)
    value nil = {n = Nat.zero;
                 l = List.nil}
    value cons = fun ~head ~tail:{n; l} ->
      {n = Nat.succ ~n;
       l = List.cons ~head ~tail:l}
```

```
    value length = fun ~l:{n; l} -> n
  end
```

1.2.3. Anonymous coinductive types

OCaml has no coinductive structured recursion operations, neither has Functorial ML, but the latter would be easy to extend. Charity has coinductive structured recursion operations and coinductive types. PolyP has operations, but they are typed using a kind of recursive types, the same used for inductive operations, so that a single type can be regarded as both inductive and coinductive. We consider recursive types unsafe (a value has many types), especially in a language with anonymous datatypes and polytypic combinators with ambiguous typing. Nevertheless, the PolyP approach seems convenient, e.g., for libraries, and we think it can be partially reconstructed using modules, see Section 3.2.

The code in this chapter is inspired by example programs written for the Charity programming language [1]. For coinductive types we provide an inverse conversion `ind2t`, since the coinductive structured recursion combinator `unfold` produces values of a coinductive type that have to be converted to the abstract type. For coinductive types based off sum types the conversion `tde` turns out to be quite useful, as well, mainly to enable case analysis over the sum type. Time constructors of lists are not mandatory, since we can construct lists with `unfold` but, especially for finite coinductive lists, they are useful.

```
spec CoList =
~Elem:sig type t end ->
  sig
    type t
    value ind2t : ~it:coind c: [‘Nil|‘Cons {head : Elem.t; tail : c}] -> t
    value tde : ~it:t -> [‘Nil|‘Cons {head : Elem.t; tail : t}]
    value nil : t
    value cons : ~head:Elem.t ~tail:t -> t
  end
module CoList =
  struct
    type t = coind c: [‘Nil|‘Cons {head : Elem.t; tail : c}]
    value ind2t = fun ~it -> it
    value tde = fun ~it -> it . unde
    value nil = ‘Nil . uncon
    value cons = fun ~head ~tail -> {head; tail} .‘Cons . uncon
  end
```

The destructor combinator `unde` taking an inductive type value to an underlying type (in this case a sum type) has an inverse `uncon`. The constructor combinator `uncon` can only construct finite values of the coinductive lists type, as opposed to `unfold` that can be used to construct infinite lists.

In the following example we implement catenation of coinductive lists. Since `tde` is available, we could implement the operation with general recursion and without `ind2t`. But then the computation fails to terminate whenever list `l1` is infinite. In such cases, out code with `ind2t` and `unfold` terminates and produces an infinite list (observationally equal to `l1`).

```
spec CoListOps =
~CoList ->
  sig
    value append : ~l1:CoList.t ~l2:CoList.t -> CoList.t
    value is_nil : ~l:CoList.t -> [‘True|‘False]
  end
module CoListOps =
```

```
  struct
    value append = fun ~l1 ~l2 ->
      CoList.ind2t ~it:
        match {l1; l2} with
        unfold {l1; l2} ->
          match CoList.tde ~it:l1 with
          [‘Nil ->
              match CoList.tde ~it:l2 with
              [‘Nil -> ‘Nil
              |‘Cons {head; tail} -> {head; tail = {l1; l2 = tail}} .‘Cons]
          |‘Cons {head; tail} -> {head; tail = {l1 = tail; l2}} .‘Cons]
    value is_nil = fun ~l ->
      match CoList.tde ~it:l with
      [‘Nil -> ‘True
      |‘Cons -> ‘False]
  end
```

Compare the code of `append` with the corresponding code in Charity [1]:

```
def coappend: colist(A) * colist(A) -> colist(A)
    = (l1, l2) => (| (ff,          ff)          =>
                         delist: ff
                   | (ff,          ss (a, l2')) =>
                         delist: ss (a, (ff, delist l2'))
                   | (ss (a, l1'), l2')         =>
                         delist: ss (a, (delist l1', l2'))
                   |)
                      (delist l1, delist l2).
```

Our code looks, arguably, somewhat more conventional, but less succinct. The conversions (three in this case) do not seem to obscure the overall pattern of recursion, but they introduce three additional lines of code. Simultaneous pattern-matching over both lists, as in the Charity code, is not yet possible in Dule, so we use nested case expressions.

Coinductive natural numbers can be defined similarly as coinductive lists. The `infinity` value represents the “infinite” natural number.

```
spec CoNat =
  sig
    type t
    value ind2t : ~it:coind c: [‘Zero|‘Succ c] -> t
    value tde : ~it:t -> [‘Zero|‘Succ t]
    value zero : t
    value succ : ~n:t -> t
    value infinity : t
  end
CoNat =
  struct
    type t = coind c: [‘Zero|‘Succ c]
    value ind2t = fun ~it -> it
    value tde = fun ~it -> it . unde
    value zero = ‘Zero . uncon
    value succ = fun ~n -> n .‘Succ . uncon
    value infinity =
      match {} with (* yes, {} suffices here *)
      unfold u -> u .‘Succ
  end
```

We can now define `length` of coinductive lists. Notice that infinite lists have infinite length (but the computation terminates).

```
spec CoLength =
~CoList ~CoNat ->
  sig
    value length : ~l:CoList.t -> CoNat.t
  end
module CoLength =
  struct
    value length = fun ~l ->
      CoNat.ind2t ~it:
        match l with
        unfold l ->
          match CoList.tde ~it:l with
          [‘Nil -> ‘Zero
          |‘Cons {head; tail} -> tail .‘Succ]
  end
```

The separation of (co)inductive and sum types allows us to take a coinductive closure of arbitrary types. The type of infinite lists (streams) below is a coinductive closure of a product type (a type of records). Here operations `head` and `tail` seem more natural than `tde`, and we stay with a conventionally looking `cons` instead of the inverse to `tde`, called `tcon` and typed `tcon : ~it:{head : Elem.t; tail : t} -> t`. Conversion `ind2t` is still indispensable.

```
spec InfList =
~Elem:sig type t end ->
  sig
    type t
    value ind2t : ~it:coind c: {head : Elem.t; tail : c} -> t
    value head : ~it:t -> Elem.t
    value tail : ~it:t -> t
    value cons : ~head:Elem.t ~tail:t -> t
  end
module InfList =
  struct
    type t = coind c: {head : Elem.t; tail : c}
    value ind2t = fun ~it -> it
    value head = fun ~it -> it . unde . head
    value tail = fun ~it -> it . unde . tail
    value cons = fun ~head ~tail -> {head; tail} . uncon
  end
```

Infinite lists type has only infinite values, so they are all constructed with `unfold` and converted with `ind2t`.

```
spec InfListOps =
~InfList ->
  sig
    value stutter : ~m:Elem.t -> InfList.t
    value alternate : ~first:Elem.t ~second:Elem.t -> InfList.t
  end
module InfListOps =
  struct
    value stutter = fun ~m ->
      InfList.ind2t ~it:
        match {} with
```

```
        unfold u -> {head = m; tail = u}
    value alternate = fun ~first ~second ->
      InfList.ind2t ~it:
        match `True with
        unfold b ->
          match b with
          [`True -> {head = first; tail = `False}
          |`False -> {head = second; tail = `True}]
  end
```

Inductive and coinductive types can coexist peacefully. Here the structured recursion is performed over an inductive natural number (hence the conversion `Nat.t2ind`) and on each pass an infinite list is beheaded.

```
spec NthInfList =
~Nat ~InfList ->
  sig
    value nth : ~n:Nat.t ~l:InfList.t -> Elem.t
  end
module NthInfList =
  struct
    value nth = fun ~n ~l ->
      let nth_tail =
        match Nat.t2ind ~it:n with
        fold [`Zero -> l
             |`Succ tl -> InfList.tail ~it:tl]
      in InfList.head ~it:nth_tail
  end
```

1.3. Polymorphism through polytypism

We have seen in Section 1.1.3 that capturing structured recursion mechanisms as ordinary functions, such as `foldf` and `foldr`, eliminates the polytypic aspect (independence of datatype structure) of structured recursion and requires polymorphism at the level of module specifications to retain the polymorphic aspect (independence of the type of data stored in the datatype). Fine-grained modularization favors monomorphic typing of operations (dependence on types is captured explicitly through dependence on modules) so we want to demonstrate that our proposed form of (co)inductive types and their exporting does not conflict with this programming style — we would like to show that polytypism of our constructions is orthogonal to polymorphism. Moreover, our examples will indicate that in the context of a monomorphic modular programming language, such as Dule, the built-in polytypic structured recursion combinators behave in a polymorphic way, though in a limited context. This limited form of polymorphism carries across module boundaries, if only modules are equipped with suitable (co)inductive conversions. This time we see that the inverse of `t2ind` conversion can be useful for inductive types, too.

1.3.1. Polymorphism in monomorphic core language

There is no polymorphism in our programming language Dule. Any dependency on types in Dule is expressed modularly. This enforces better awareness of multiple type instantiations of entities appearing in the program, usually making the interfaces narrower and proving polymorphism unnecessary for this particular case. Another result can be the early elimination of potential bugs, when the more strict module dependency mechanism explicitly imposes requirements on the types to be used in instantiations. For example, if an ordering on a type is needed only in a rarely used operation, other polymorphic operations of the same module can be successfully applied to values without ordering, leading to a ground-up rewrite when the ordering is finally found necessary.

However, the monomorphic nature of the language does not limit the generality of the built-in polytypic structured recursion combinators. Consider the following two values, one of the type of lists of natural numbers, the other of the type of lists of boolean values.

```
let intlist = {h = Nat.zero;
               t = {h = Nat.succ ~n:Nat.zero;
                    t = ‘Nil . con} .‘Cons . con} .‘Cons . con
    boollist = {h = ‘True;
                t = {h = ‘False;
                     t = ‘Nil . con} .‘Cons . con} .‘Cons . con
```

In the same fragment of code we can use the built-in structured recursion combinators for datatypes not only with different structure, but also with different types of data. This is illustrated with by the three uses of `fold` below.

```
in
let lenght_of_intlist = match intlist with
                        fold [‘Nil -> Nat.zero
                             |‘Cons {h; t} -> Nat.succ ~n:t]
    lenght_of_boollist = match boollist with
                         fold [‘Nil -> Nat.zero
                              |‘Cons {h; t} -> Nat.succ ~n:t]
    add = fun ~n ~it ->
      match Nat.t2ind ~it with
      fold [‘Zero -> n
           |‘Succ m -> Nat.succ ~n:m]
in
add ~n:lenght_of_intlist ~it:lenght_of_boollist
```

This recovery of polymorphism through polytypism, its limitations and interaction with a module system, are better illustrated with our built-in combinator `map`. After introducing the combinator we will return to examples.

1.3.2. Mapping

The keyword `map` signals the operation of traversing a complex value, applying the given function to sub-values on the way. This is a very broad generalization of the mapping function for lists, well known in functional programming. The mapping function for lists, operating with a function `f` may be recast in Dule as simply `map f`. In the following example we obtain a two-element list containing the falsity value and the truth value.

```
let list_map = fun ~f ~l ->
  match l with
  map f
in
let l = {h = ‘True;
         t = {h = ‘False;
              t = ‘Nil . con} .‘Cons . con} .‘Cons . con
in
list_map ~l ~f:[‘True -> ‘False
                |‘False -> ‘True]
```

The combinator `map` operates with functions with parameters labeled `it`. Any case analysis expression is such a function, hence the application of `list_map` is type-correct.

A mapping on binary trees operating with a function `f` is written in the same way: `map f`. In the following example we construct a tree and then apply the mapping combinator, obtaining a tree with a single value `‘True`. Function `Nat.is_zero` accepts its arguments at label `it`, just as `map`

requires.

```
let tree_map = fun ~f ~t ->
  match t with
  map f
in
let t = {valu = Nat.zero;
         left = ‘Empty . con;
         right = ‘Empty . con} .‘Node . con
in
tree_map ~f:Nat.is_zero ~t
```

Notice that `list_map` and `tree_map` have the same code — `map` is a polytypic combinator.

One can map not only over values of inductive types, but over a value of every type (except, in general, a function type — the details are out of scope of this paper). Below we define a function negating every value on a coinductive stream (an infinite list, see Section 1.2.3). Then the function is tested and the resulting stream is destructed to obtain its first element, which turns out to be `‘True`.

```
let negate_stream = map [‘True -> ‘False
                        |‘False -> ‘True]
    falsities = match {} with
                unfold tail -> {head = ‘False; tail}
in
let truths = negate_stream ~it:falsities in
truths . unde . head
```

The expression below checks and notes whether a natural number is zero, for every natural number value contained in the record:

```
match {b = ‘False; n = Nat.zero;
       r = {rn = Nat.succ ~n:Nat.zero; ro = ‘OK}} with
map Nat.is_zero
```

The result will be a similar record but with `‘True` in place of `Nat.zero` and `‘False` in place of `Nat.succ ~n:Nat.zero`. The type-checker will reconstruct such a type for the `map` combinator so that the application of the combinator will retain the old values at the `b` and `ro` fields of the record. In the following example we increment three natural numbers. We write numbers in decimal notation, with the use of a syntactic sugar and create the functional argument to `map` on the spot, with another sugared notation, the same that was used for `unfold` before.

```
let incr =
  (map n -> Nat.succ ~n)
in
{n1n2 = incr ~it:({n1 = 5; n2 = 7; b = ‘True} .‘Two);
 n = incr ~it:(13 .‘One)}
```

In the resulting record there will be numbers 6, 8 and 14, respectively.

The astute reader may ask “what if I want only one of the numbers `n1`, `n2` above to be incremented or, in another application, only attributes of an AVL tree, and not its elements?”. This could be answered by providing an additional syntax for directing the type reconstruction for mappings more precisely. Instead we suggest that the desired functions be written by hand, using case analysis, iteration, etc. Some remarks about how typing determines sub-values to be changed and a discussion of the lack of principal typing for mapping is provided in Section 2.4.

1.3.3. Polymorphism with mapping

Let us revisit our lists of integers and of boolean values, this time using the polytypic combinator `map` to operate over the lists in a polymorphic fashion.

```
let intlist = {h = 0;
               t = {h = 1;
                    t = ‘Nil . con} .‘Cons . con} .‘Cons . con
    boollist = {h = ‘True;
                t = {h = ‘False;
                     t = ‘Nil . con} .‘Cons . con} .‘Cons . con
    monomorphic_map =
      map fun ~it -> {x = it; y = it}
in
  {intmap =
     match intlist with
     map fun ~it -> {x = it; y = it}
  ;boolmap =
     match boollist with
     map fun ~it -> {x = it; y = it}
  ;intmap2 =
     match intlist with
     monomorphic_map
(* this would be incorrect:
  ;boolmap2 =
     match boollist with
     monomorphic_map *)
  }
```

If `map` was not polymorphic, through its inherent polytypism, different versions would have to be used for integer lists and boolean lists. Note, however, that as soon as the combinator is named as a language value (as with `monomorphic_map` above) it becomes monomorphic. In our monomorphic programming language the exact monomorphic type is not determined until a value is used (as `monomorphic_map` under `intmap2` above), but any use fixes the type. Consequently, two different uses of the named mapping can lead to inconsistent typings, as would be the case with `boolmap2` in the example above.

Unfortunately, not only `map` has its type fixed upon being named — this is also the case with arguments to `map`, so that `boolmap` in the following piece of code may be incorrect, similarly as `boolmap2` in the example above.

```
let monomorphic_delta =
      fun ~it -> {x = it; y = it}
in
(* if [intmap] and [boolmap] are lists of records then here typing fails *)
  {intmap =
     match intlist with
     map monomorphic_delta
  ;boolmap =
     match boollist with
     map monomorphic_delta
  }
```

This implies that there is not way to avoid spurious copying of code, when using a polytypic combinator with the same argument in a polymorphic way. However, as long as a combinator is used in a polytypic, but monomorphic way, as in the following example, no copying of argument code is necessary.

```
let intlist = {h = 0;
               t = {h = 1;
                    t = ‘Nil . con} .‘Cons . con} .‘Cons . con
    inttree = {valu = 5;
               left = ‘Empty . con;
               right = ‘Empty . con} .‘Node . con
    monomorphic_delta =
      fun ~it -> {x = it; y = it}
in
  {listmap =
     match intlist with
     map monomorphic_delta
  ;treemap =
     match inttree with
     map monomorphic_delta}
```

Concluding, Dule built-in polytypic combinators behave in a polymorphic way, but they arguments do not, so if the arguments are large and similar, but differently typed, the polymorphic code-reuse fails and modules have to be used.

1.3.4. Polymorphism through monomorphic modules

Recall our specification of a module of lists with elements of type `Elem.t`. We enrich the specification by requiring module `Elem` to provide an action on type `Elem.t`. We also add a conversion `ind2t`, the inverse of `t2ind`, to the specification of lists.

```
spec List =
~Elem:sig
        type t
        value action : ~it:t -> t
      end ->
  sig
    type t
    value t2ind : ~it:t -> ind list: [‘Nil|‘Cons {head : Elem.t; tail : list}]
    value ind2t : ~it:ind list: [‘Nil|‘Cons {head : Elem.t; tail : list}] -> t
    value nil : t
    value cons : ~head:Elem.t ~tail:t -> t
  end
```

In the context of this specification we can reconstruct the behavior of `map` from the previous examples, where it was used independently of the type of data stored in lists — in a polymorphic way. As an example we implement function `traverse`, that applies `Elem.action` to each element of a list. The code of the function is independent of the type `Elem.t`, due to the generality of `map`.

```
spec MapAction =
~List ->
  sig
    value traverse : ~it:List.t -> List.t
  end
module MapAction =
  struct
    value traverse = fun ~it ->
      List.ind2t ~it:
        match List.t2ind ~it with
        map Elem.action
  end
```

Without the use of `List.ind2t` the code would be incorrect, because the result of `map` is an inductive type, not an abstract type, as specification of `traverse` requires. Inside module `MapAction` the two types are not equal, even if `List.t` is indeed implemented as an inductive type. The conversion `List.ind2t` can be, in simplest cases, implemented as an identity, just as `List.t2ind`. If the `List.ind2t` conversion was not provided in `List`, it would have to be substituted by a use of `fold`, as in the following alternative implementation of `traverse`. A complete `fold` expression is function with arguments at `it` (just as a case expression, an `unfold` expression and a mapping expression), hence the application at label `it`.

```
value traverse = fun ~it ->
  (fold [‘Nil -> List.nil
        |‘Cons {head; tail} -> List.cons ~head ~tail])
  ~it:
    match List.t2ind ~it with
    map Elem.action
```

Conversion `List.ind2t` is always expected to give the same results as a `fold` expression analogous to the one above, even if the abstract type is not implemented as the inductive type from the typing of `List.ind2t`.

1.4. Summary

1.4.1. Four conversions

We have demonstrated the use of anonymous inductive and coinductive types separated from sum and product types, their related built-in combinators and four conversion operations `t2ind`, `ind2t`, `tde`, `tcon`. The conversion operations for standard lists are typed as in the following specification. After the specification we also present the implementation in the simplest case — where the abstract type is implemented just as the inductive type seen in the typing of operations.

```
spec List4 =
~Elem:sig type t end ->
  sig
    type t
    value t2ind : ~it:t -> ind list: [‘Nil|‘Cons {head : Elem.t; tail : list}]
    value ind2t : ~it:ind list: [‘Nil|‘Cons {head : Elem.t; tail : list}] -> t
    value tde : ~it:t -> [‘Nil|‘Cons {head : Elem.t; tail : t}]
    value tcon : ~it:[‘Nil|‘Cons {head : Elem.t; tail : t}] -> t
  end
module List4 =
  struct
    type t = ind list: [‘Nil|‘Cons {head : Elem.t; tail : list}]
    value t2ind = fun ~it -> it
    value ind2t = fun ~it -> it
    value tde = fun ~it -> it . de
    value tcon = fun ~it -> it . con
  end
```

In our earlier specification of lists, instead of `tcon` we have provided the more convenient constructor operations `nil` and `cons`. Conversion `tcon` can be recovered from them in the following way.

```
value tcon = fun ~it ->
  match it with
    [‘Nil -> List.nil
    |‘Cons {head; tail} -> List.cons ~head ~tail]
```

For infinite lists — streams implemented earlier in module `InfList` — we have the following typing of the four conversions. Notice that the simplest implementation of operations is completely analogous to that for standard lists.

```
spec InfList4 =
~Elem:sig type t end ->
  sig
    type t
    value t2ind : ~it:t -> coind c: {head : Elem.t; tail : c}
    value ind2t : ~it:coind c: {head : Elem.t; tail : c} -> t
    value tde : ~it:t -> {head : Elem.t; tail : t}
    value tcon : ~it:{head : Elem.t; tail : t} -> t
  end
module InfList4 =
  struct
    type t = coind c: {head : Elem.t; tail : c}
    value t2ind = fun ~it -> it
    value ind2t = fun ~it -> it
    value tde = fun ~it -> it . unde
    value tcon = fun ~it -> it . uncon
  end
```

The constructor of our earlier implementation of streams is almost identical to `tcon`.

```
    value tcon = fun ~it:{head; tail} -> InfList.cons ~head ~tail
```

The destructors are also close to `tde`.

```
    value tde = fun ~it -> {head = InfList.head ~it; tail = InfList.tail ~it}
```

1.4.2. Two conversions suffice

We have argued that for inductive types, the most important operations are `t2ind` and `tcon`, while for coinductive types they are `ind2t` and `tde`. We have also indicated some of the cases, where the remaining two operations are useful. Both pairs of operations are enough to express all four but, especially in case of straightforward implementation of the abstract type, this can be relatively costly.

In case of standard lists, when we have `t2ind` and `tcon` we can express `ind2t` in the following way, already used in an alternative implementation of `traverse` in the previous section (remember that a `fold` expression is a function).

```
    value ind2t =
      fold [‘Nil -> List.nil
           |‘Cons {head; tail} -> List.cons ~head ~tail]
```

This is equivalent to

```
    value ind2t =
      fold [‘Nil -> tcon ~it:‘Nil
           |‘Cons ht -> tcon ~it:ht .‘Cons]
```

which is equivalent to

```
    value ind2t = fold tcon
```

We can reconstruct `tde` in the following way.

```
    value tde = fun ~it ->
```

```
  match t2ind ~it with
  fold [‘Nil -> ‘Nil
       |‘Cons {head; tail} ->
          {head; tail =tcon ~it:tail} .‘Cons]
```

And this is equivalent to

```
value tde = fun ~it ->
  match t2ind ~it with
  fold map tcon
```

Notice that we have managed to express the two remaining operations in a polytypic way — without mentioning the type structure of lists. In case of coinductive types with conversions `ind2t` and `tde` the construction is dual, as follows.

```
value t2ind = unfold tde

value tcon = fun ~it ->
  ind2t ~it:
    match it with
    unfold map tde
```

Fact 1.1. *In the simplest case, where the implementation of an abstract type coincides with the (co)inductive type of its specification,* `t2ind` *and* `tcon` *for inductive types and* `ind2t` *and* `tde` *for coinductive types suffice to express all four conversion operations, so that they have the same semantics as in the straightforward implementation.*

Proof. Diagram chasing using the semantics described in Section 2. □

It is easy to see that to reconstruct the four conversions using structured recursion one needs `t2ind` with either `ind2t` or `tcon`, for inductive types, and `ind2t` with `t2ind` or `tde`, for coinductive types. With general recursion any pair with different domains and codomains suffices.

2. SEMANTICS AND TYPING

2.1. Semantics

A strict account of the syntax, typing and semantics of our programming language Dule, together with a discussion of variants and proofs of properties is given in [9]. The mathematical model of the core language of Dule is an (almost) 2-category [5] with products and some other additional structure. The category is not strictly a 2-category because the interchange law not always holds. However, a programmer, as well as, for the most part, a designer of a module system can safely regard the underlying model as a 2-category, or even just $\mathbf{Cat}$ — the category of all (small) categories. Objects of the 2-category model kinds of the programming language, 1-morphisms model types and 2-morphisms model values. The kind of ordinary types can be though of as the object $\mathbf{Set}$ in $\mathbf{Cat}$ — the category of sets and functions. Then types become sets and values are functions. To avoid the compilations of variance analysis we declare undefined any mapping combinators and (co)inductive operations involving types where “recursion” goes through the exponent construction. Fortunately inductive types with the induction parameter nested inside a function type are not very common in programming practice.

As is customary for categories, the typing of our core language assigns to a value not only the codomain type but also the domain type. The domain type models the types of values in the environment. In this setting we model value variables as projections in the category. The composition of values written with a dot in the programming language is modelled as the vertical composition of 2-morphisms in the category. Sum type constructors are

coproduct injections, inductive type constructors are constructor morphisms of initial algebras, etc. There is a lot of substitution notation in the typing rules for Dule values. For example, $g[h/i]$ denotes instantiation of type g with type h in place of the type variable i. If $g = $ `[‘Nil|‘Cons {head : Elem.t; tail : list}]`, then the inductive type of lists is equal to `ind list:`g and g[`List.t/list`] is exactly the codomain type of the conversion `tde`. However, there is no syntactic substitution in our language. The composition in the underlying category faithfully and semantically implements the type substitution or, in other words, the language of types is referentially transparent.

2.2. Syntactic sugar

We design a uniform way of supplying operands to coproduct, (co)inductive and mapping combinators. Arguments to combinators in all of the rules (5), (6), (8) and (11) below are typed in the same way: as functions with one of the parameters called `it`. The results of the combinators are also typed in this way so that using the results of nested combinators as arguments to others is easy. Consequently the most frequently occurring kind of application in Dule is that with a single label `it`. For these cases there is an additional syntactic form for application, written using keywords `match` and `with`, which denotes an application of a single argument at label `it`. For example

```
(unfold it -> ‘True) ~it:5
```

is equivalent to

```
match 5 with (unfold fun ~it -> ‘True)
```

The syntax is especially handy for delimiting large arguments, as in the example below.

```
match (Nat.pred ~n:(Nat.add ~n:1 ~it:2)) . de with
[‘Zero -> ‘True
|‘Succ -> ‘False]
```

Longer and more realistic examples can be found throughout the paper.

2.3. Typing rules

The Dule core language is strongly and statically typed. We will denote the typings by “$\triangleright$” not by “ : ” to avoid confusion with the colons of the language constructions. For brevity we will omit the rules assigning kinds to types and present only some of the rules for derivation of domain and codomain types of values.

$$\frac{t \triangleright f \to g \qquad u \triangleright g \to h}{t\ .\ u \triangleright f \to h} \tag{1}$$

$$\frac{}{i \triangleright \{i : f;\ \ldots\} \to f} \tag{2}$$

$$\frac{t_1 \triangleright f \to h_1 \quad \cdots \quad t_n \triangleright f \to h_n}{\{i_1 = t_1;\ \ldots;\ i_n = t_n\} \triangleright f \to \{i_1 : h_1;\ \ldots;\ i_n : h_n\}} \tag{3}$$

$$\frac{}{‘i \triangleright f \to [‘i\ \ f|\ \ldots]} \tag{4}$$

$$\frac{\begin{array}{c} t_1 \triangleright f \to \texttt{\~{}it:}f_1\ \texttt{\~{}}j_1\texttt{:}g_1\ \ldots\ \texttt{\~{}}j_n\texttt{:}g_n\ \texttt{->}\ h \\ t_2 \triangleright f \to \texttt{\~{}it:}f_2\ \texttt{\~{}}j_1\texttt{:}g_1\ \ldots\ \texttt{\~{}}j_n\texttt{:}g_n\ \texttt{->}\ h \\ \vdots \end{array}}{\begin{array}{c} \texttt{[`}i_1\ \ t_1\texttt{|`}i_2\ \ t_2\texttt{|}\ \ldots\texttt{]} \triangleright f \to \\ \texttt{\~{}it:[`}i_1\ \ f_1\texttt{|`}i_2\ \ f_2\texttt{|}\ \ldots\texttt{]}\ \texttt{\~{}}j_1\texttt{:}g_1\ \ldots\ \texttt{\~{}}j_n\texttt{:}g_n\ \texttt{->}\ h \end{array}} \tag{5}$$

$$\frac{t \triangleright f \to \texttt{\~{}it:}g'\ \texttt{\~{}}j_1\texttt{:}g_1\ \ldots\ \texttt{\~{}}j_n\texttt{:}g_n\ \texttt{->}\ h}{\texttt{map}\ t \triangleright f \to \texttt{\~{}it:}g[g'/i]\ \texttt{\~{}}j_1\texttt{:}g_1\ \ldots\ \texttt{\~{}}j_n\texttt{:}g_n\ \texttt{->}\ g[h/i]} \tag{6}$$

$$\frac{}{\texttt{con} \triangleright g[\texttt{ind}\ i\texttt{:}\ g/i] \to \texttt{ind}\ i\texttt{:}\ g} \tag{7}$$

$$\frac{t \triangleright f \to \texttt{\~{}it:}g[h/i]\ \texttt{\~{}}j_1\texttt{:}g_1\ \ldots\ \texttt{\~{}}j_n\texttt{:}g_n\ \texttt{->}\ h}{\texttt{fold}\ t \triangleright f \to \texttt{\~{}it:(ind}\ i\texttt{:}\ g\texttt{)}\ \texttt{\~{}}j_1\texttt{:}g_1\ \ldots\ \texttt{\~{}}j_n\texttt{:}g_n\ \texttt{->}\ h} \tag{8}$$

$$\frac{}{\texttt{de} \triangleright \texttt{ind}\ i\texttt{:}\ g \to g[\texttt{ind}\ i\texttt{:}\ g/i]} \tag{9}$$

$$\frac{}{\texttt{uncon} \triangleright g[\texttt{coind}\ i\texttt{:}\ g/i] \to \texttt{coind}\ i\texttt{:}\ g} \tag{10}$$

$$\frac{t \triangleright f \to \texttt{\~{}it:}h\ \texttt{\~{}}j_1\texttt{:}g_1\ \ldots\ \texttt{\~{}}j_n\texttt{:}g_n\ \texttt{->}\ g[h/i]}{\texttt{unfold}\ t \triangleright f \to \texttt{\~{}it:}h\ \texttt{\~{}}j_1\texttt{:}g_1\ \ldots\ \texttt{\~{}}j_n\texttt{:}g_n\ \texttt{->}\ \texttt{coind}\ i\texttt{:}\ g} \tag{11}$$

$$\frac{}{\texttt{unde} \triangleright \texttt{coind}\ i\texttt{:}\ g \to g[\texttt{coind}\ i\texttt{:}\ g/i]} \tag{12}$$

2.4. Properties

The reconstruction of domain and codomain types of a given value is very difficult. A portion of the difficulty comes from the complicated two-part unification procedure, reflecting the additional way a type may be more general than the other — by having less component in some of the indexed lists of its arguments. Another difficulty comes from the absence of a fixed "environment". Still another complication comes from the polytypic combinators and their interaction with unification of indexed lists. The most problematic combinator seems to be the mapping combinator, as captured by rule (6), where type g is hard to guess on the basis of its two occurrences with substituted subterms (in particular if the subterms are not proper!). The definition of generality of typing that seems closest to the similar definitions for conventional type systems is the following.

Definition 2.1. *An indexed list of types lf is said to be less or equally detailed than lg, if for each element f of lf at index k, the element k of lg exists and is less or equally detailed than f. A type term f is less or equally detailed than g if the semantics of f and g are equal or f is the empty product type* `{}` *or f and g have the same root constructor and the corresponding type or type list subterms of f are less or equally detailed than the corresponding subterms of g.*

Fact 2.1. *There are typable values that have no least detailed elements in their sets of derivable domains nor in their sets of derivable codomains. For example the term*

```
(map fun ~(it:[‘A]) -> ‘B) ~it:‘A
```

can have both `[‘A]` *and* `[‘B]` *as its codomain, but cannot have* `{}` *(their only lower bound).*

Our type-reconstruction algorithm finds a minimally detailed typing for a given value. Choosing which of the minimally detailed typings are to be generated by the compiler will be an interesting research topic, most probably involving case studies. We believe that with a good choice of typing defaults in the compiler, the programmer will very rarely have to direct type-reconstruction by explicit typing, especially that our language encourages fine-grained modularization, already providing numerous typing hints.

The typing and semantics of Dule enjoys the following soundness properties.

Fact 2.2. *The typing of values is compatible with domains and codomains of their semantics in the category.*

Fact 2.3. *The inferred type of every value is expressible, moreover expressible an anonymous (closed) type expression.*

Fact 2.4. *The language of Dule values is referentially transparent with respect to value projections (programming language value variables).*

3. CONCLUSION AND FUTURE WORK

3.1. Conclusion

W have proposed language constructs that allow polytypic combinators to be used across abstract module boundaries. The constructs are the anonymous inductive and coinductive types together with their accompanying combinators that are used both to export conversion operations in module interfaces and to perform structured recursion over abstract types. In addition to being anonymous, our (co)inductive types are also separated from sum and product types, which facilitates a stricter typing of conversion operations. The constructs are parts of an implemented modular programming language Dule, but their analogues can be found in other languages. The described modular programming discipline can be useful, for some applications, even in languages with no polytypic operations.

Here is a summary of advantages and drawbacks of our approach, as discussed in the paper.

Advantages:

- absolute abstraction
- but for a particular implementation of a datatype, one can optimize both the code of the conversions and the choice of conversion to be used
- in the simplest case, full power, as if there was no modules; but also the ability to exchange implementations
- pattern matching
- types of conversions indicate if a datatype is meant for structured or general recursion and provide some safety against erroneous usage; the set of conversions may be different in different specifications of the same module
- very explicit but relatively concise notation for programmers; separation of (co)inductive, sum and product types eliminates dummy injections and projections

- admits full polytypism with at most four conversions, regardless of the number of polytypic combinators that are to operate on the datatype
- does not require type abbreviations and polymorphism in module specifications
- polytypism of combinators is orthogonal to polymorphism; no polymorphism required at the core language level and, to an extent, the polymorphism can be simulated through polytypism
- simple semantics of types and uniform typing of combinators
- experiments with this modular programming style in OCaml confirm its usefulness for grammar datatypes with additional structure, such as stamps for hash-consing or source code locations

Drawbacks:

- verbose and exotic
- the "simplest implementation is the best specification" approach is not always valid
- for a single datatype only inductive combinators allowed or only coinductive combinators; hence restricted patterns of structured recursion
- no principal typing for `map`
- experiments in OCaml reveal that nested pattern matching (`Cons(Zero, Nil) -> ...`) is not possible and the resulting nested conversions `tde` are unreadable

3.2. Future work

Our constructs have much in common with views [15], designed to facilitate pattern matching in the presence of abstract types at the core language level. Anonymous (co)inductive types are more general in that they enable not only pattern matching, but also structured recursion. Views are more general in that they not always choose the simplest specification (view) of a datatype, admitting many useful views of a single implementation, instead of our many implementations of a single canonical specification. We expect our canonical implementation with its four conversions to be convenient for defining other views as modules with the canonical specification as their domain. Here is a sketch of a module corresponding to a view from [15].

```
spec NatEvenOdd =
~Nat ->
  sig
    value t2view : ~it:Nat.t -> ind t: [‘Zero|‘Even t|‘Odd t]
    value view2t : ~it:ind t: [‘Zero|‘Even t|‘Odd t] -> Nat.t
  end
NatEvenOdd =
  struct
    value t2view = fun ~it ->
      match Nat.t2ind ~it with ...
    value view2t =
      fold [‘Zero -> Nat.zero
           |‘Even n -> Nat.mult ~n ~it:2
           |‘Odd n -> Nat.add ~n:(Nat.mult ~n ~it:2) ~it:1]
  end
```

The modular notation with a light-weight module system should not be much more burdensome than the views notation, at the same time providing systematization and even greater abstraction. More experiments are needed.

An interesting concept and methodology is the merger of operations of algebras and co-algebras sharing the same carrier [2]. A generalization of a variant of this concept is already expressible in Dule by defining a co-algebra inside a module and providing both co-algebraic and algebraic conversion operations for the carrier. For example we could provide the following operations for lists.

```
spec BothList =
~Elem:sig type t end ->
  sig
    type t
    value t2ind : ~it:t -> ind list: [‘Nil|‘Cons {head : Elem.t; tail : list}]
    value ind2t : ~it:coind c: [‘Nil|‘Cons {head : Elem.t; tail : c}] -> t
    value tde : ~it:t -> [‘Nil|‘Cons {head : Elem.t; tail : t}]
    value nil : t
    value cons : ~head:Elem.t ~tail:t -> t
  end
```

Unfortunately, here the implementation of `t2ind` is complex, costly and not always terminating. Experiments are needed to find out in which cases such programming discipline is practical and if we should extend the language and categorical model in order to increase the usability and efficiency of compilation for this construction. Even if the underlying implementation is eventually reduced to recursive types, this is still safer than PolyP, because the type conversions are explicit and so a single abstract value has a single type.

The `unwrap` operation of two-level types [14] directly corresponds to our conversion `tde`, though similarly as in PolyP, the lack of separation between inductive and sum types in the underlying Haskell type system results in a sum-inductive type with void closure and a dummy constructor. However, the sum-type level of two-level types is not the same as the sum type in the codomain of `tde` — it is a parameterized type. Consequently, in Dule it can only be expressed with a module parameterized by type — and such modules are a companion topic to two-level types, appearing together in example applications [14]. In Dule it is possible to inductively close such a module, and even more, we can close a set of mutually dependent modules that depend not only on types, but on each other. It would be interesting to recast two-level types' examples in Dule and see if the dependency on whole specifications, instead of types, affords any improvements in modularization and if the modular and (co)inductive notation scales for such levels of abstraction as demonstrated with the two-level types methodology.

REFERENCES

[1] Robin Cockett. Charitable Thoughts, 1996. (draft lecture notes, `http://pll.cpsc.ucalgary.ca/charity1/www/home.html`).

[2] Martin Erwig. Categorical programming with abstract data types. In Armando Martin Haeberer, editor, *AMAST*, volume 1548 of *Lecture Notes in Computer Science*, pages 406–421. Springer, 1998.

[3] Tom Fukushima and Charles Tuckey. *Charity User Manual*, January 1996. (draft, `http://pll.cpsc.ucalgary.ca/charity1/www/home.html`).

[4] P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[5] C. B. Jay. An introduction to categories in computing. Technical Report UTS-SOCS-93.9, University of Technology, Sydney, 1993.

[6] C. B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.

[7] Johan Jeuring and Patrik Jansson. Polytypic Programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Tutorial Text from 2nd Int. School on Advanced Functional Programming, Olympia, WA, USA, 26–30 Aug 1996*, volume 1129 of *Lecture Notes in Computer Science*, pages 68–114. Springer-Verlag, Berlin, 1996.

[8] Mikołaj Konarski. Source code of the Dule compiler. `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd`, 2005.

[9] Mikołaj Konarski. Application of category-theory methods to the design of a system of modules for a functional programming language. `http://www.mimuw.edu.pl/~mikon/Dule/download/phd-thesis/dule.pdf`, 2006.

[10] Xavier Leroy. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from `http://caml.inria.fr`.

[11] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1989.

[12] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[13] Simon Peyton Jones. Special issue: Haskell 98 language and libraries. *Journal of Functional Programming*, 13, January 2003.

[14] Tim Sheard and Emir Pasalic. Two-level types and parameterized modules. *J. Funct. Program*, 14(5):547–587, 2004.

[15] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In Steve Munchnik, editor, *Proceedings, 14th Symposium on Principles of Programming Languages*, pages 307–312. Association for Computing Machinery, 1987.