Transformations for Generating Type Refinements*

Douglas R. Smith and Stephen J. Westfold

Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304 USA
{smith,westfold}@kestrel.edu
10 April 2019

Abstract. We present transformations for incrementally defining both inductive sum/variant types and coinductive product/record types in a formal refinement setting. Inductive types are built by incrementally accumulating constructors. Coinductive types are built by incrementally accumulating observers. In each case, when the developer decides that the constructor (resp. observer) set is complete, a transformation is applied that generates a canonical definition for the type. It also generates definitions for functions that have been characterized in terms of patterns over the constructors (resp. copatterns over the observers). Functions that input a possibly-recursive sum/variant type are defined inductively via patterns on the input data. Dually, functions that output a possibly-recursive record type are defined coinductively via copatterns on the function's output. The transformations have been implemented in the Specware system [5] and have been used extensively in the automated synthesis of concurrent garbage collection algorithms [10, 13] and families of protocol-processing codes for distributed vehicle control [6].

1 Introduction

We address the problem of incrementally defining types and their operators. Rather than work in the context of a programming language, where expressions are intended to have a single precise meaning, we work in a specification and refinement setting, where a specification denotes a set of possible models or implementations that satisfy a set of constraints. Incremental development by refinement can allow a more natural staged introduction of design commitments in a formal derivation. For example, program families are naturally expressed as a refinement tree where each branch defines a distinct subfamily of programs. A natural way to express such family trees is via the incremental accumulation of constraints on the types, functions, procedures, components, and other system structure. A type may have alternative elaborations in the various branches of the family tree. A similar pattern is seen in product lines of systems and the class hierarchies of object-oriented languages.

The development of correct-by-construction code via a formal refinement process has the abstract derivation form $S_0 \longrightarrow S_1 \longrightarrow ... \longrightarrow S_n \rightsquigarrow Code$. A derivation process starts with a specification S_0 of the requirements on a desired software artifact. Each $S_i,\ i = 0, 1, ..., n$ represents a structured specification and the arrows $\rightarrow$ are refinements. The refinement from S_i to S_{i+1} embodies a design decision which narrows down the number of possible implementations. In our approach,

* This work has been sponsored in part by DARPA under agreements FA8750-10-C-0241 (CRASH) and FA8750-12-C-0257 (HACMS).

most refinement steps are generated (semi)automatically by specification transformations. The final step translates the lowest-level specification S_n to code in a suitable programming language. Semantically the effect is to narrow down the set of possible implementations of S_0 to just one, so specification refinement can be viewed as a constructive process for proving the existence of an implementation of specification S_0; i.e. proving its consistency.

We are interested in specification transformations that generate refinements together with machine-checkable proofs [12]. If a formal derivation is generated by a sequence of such refinement+proof-generating transformations, then we can chain the resulting proofs together to get a proof that the final generated specification is a correct refinement of the initial requirement-level specification. Here, we introduce transformations for incrementally defining both (1) inductive sum/variant-types and functions inductively defined on them, and (2) coinductive product/record-types and functions that are coinductively defined to produce them.

Inductive types are characterized by their constructors. In a refinement setting, we can introduce a type symbol, say T, for an intended inductive type in a specification, with some of its constructors, and without a definition. A function f that takes a T input can be characterized by axioms that specify how f acts on the existing constructors. A *pattern-based* or *constructor-based* characterization of function $f : T \to A$ with respect to constructor c is an axiom that essentially has the form $f \circ c = e$ for some well-defined expression e (e.g. see Figure 1). In subsequent refinements, we add other constructors, and add pattern-based axioms for f. At each stage in the derivation (i.e. at an intermediate specification), the models of T include a set for T defined by just the current set of constructors, and well as models that allow other constructors, and even models that are not inductive. At some point in the derivation, the developer decides that the constructor set is complete by applying a transformation, called CompleteSumType, that gives a canonical definition of T as a sum/named-variant type with just the current set of constructors. It also generates inductive definitions for functions that have been characterized by pattern-based axioms.

Dually, *coinductive types* are characterized by their observers – all that can be known about an element of the type is given by various observations of it. In a refinement setting, we can introduce a type symbol T for an intended coinductive type (cotype) in a specification, along with some of its observers. A function f that produces a T value can be characterized by axioms that specify observations of its output. A *copattern-based* or *observer-based* characterization of function $f : A \to T$ with respect to observer p is an axiom that essentially has the form $p \circ f = e$ for some well-defined expression e (e.g. see Figure 6). In subsequent refinements, we add other observers, and add appropriate copattern-based axioms that specify the output of f. At each stage in the derivation, the models of T include a set for T with just the current set of observers, and well as models that allow other observers, and even models that are not coinductive. At some point in the derivation, we declare that the observer set is complete by applying a transformation, called CompleteProductType, that gives a canonical definition of T as a product/record type with just the current set of observers as projections/fields. It also generates coinductive definitions for functions that have been characterized by copattern-based axioms.

A variety of examples illustrate these transformations. Although our techniques are applied in a purely logical/functional setting, we show how to use the transformations to develop mutable global states and heap-allocated mutable types for targeting imperative and object-oriented programming languages.

2 Basic Concepts

We present basic concepts of the formal specification-and-refinement approach used in our Specware system [5, 14]. A *specification* defines a language and constrains its possible meanings via axioms. A specification is given by a finite collection of type symbols (optionally including a definition), function symbols and their signature (optionally including a definition), and axioms over the type and function symbols. We treat predicates as Boolean-valued functions. For purposes of this paper, we focus on first-order specifications (i.e. functions do not take functions as arguments), although Specware allows higher-order specifications. The deductive closure of the axioms is a theory, so a specification is a finite presentation of a theory. Let *Spec* denote the type of specifications.

A refinement can be expressed formally via a *specification morphism* which translates the language of one specification into the language of another specification in a way that preserves theorems. Formally, a *signature morphism* from specification $S0$ to specification $S1$ is a type-consistent map from the vocabulary of $S0$ (i.e. its type and function symbols) to the vocabulary of $S1$. A *specification morphism* from $S0$ to $S1$ is a signature morphism that preserves theorems; i.e. that translates each theorem of $S0$ to a theorem of $S1$. To establish a specification morphism, it is sufficient to prove that each axiom of $S0$ translates to a theorem of $S1$. Let $Morphism$ denote the type of specification morphisms (or simply morphisms).

Specification $S1$ is an *extension* of specification $S0$ if there is an specification morphism $S0 \to S1$ whose underlying signature morphism is injective. We use importation (with possible renaming) to express extension, allowing the construction of complex specifications. More generally, specifications and their morphisms constitute a category that has colimits, which provide a general means for constructing complex specifications. A *pushout* is a special case of a colimit that we will use frequently. The pushout of two morphisms with a common domain specification $B \xleftarrow{i} A \xrightarrow{j} C$ is another pair of morphisms with a common codomain, $B \xrightarrow{j'} D \xleftarrow{i'} C$, called a *cocone*, where D is the pushout specification. Intuitively, D is the simplest specification that combines B and C modulo the common structure of A [14].

As models of specification S, we admit any structure of sets and functions that interprets at least each type and function symbol in S and that satisfies the function signatures and the axioms. This loose semantics allows structures for extensions of S to be models of S. The denotation of a specification morphism m is a map from models of the codomain of m into models of the domain – every model of $S1$ is mapped to some model of $S0$.

Specification $S0$ *refines to* $S1$ if there is a specification morphism $m : S0 \to S1$. We refer to m as a refinement and a morphism, and in context, $S1$ as a refinement of $S0$. In this paper we are interested in transformations that (semi)automatically generate refinements. A *specification transformation* is a partial function on specifications that generates a refinement: $t : Spec \to Morphism$. That is, if $t(S) = m$, then $m : S \to codomain(m)$ is a refinement of S.

An extension $e : S0 \to S1$ is *conservative* if every theorem of $S1$ that is expressed over the language of $S0$, is also a theorem of $S0$. A specification morphism is *consistent* if it preserves consistency – whenever the source specification is consistent (has a nonempty set of models), then the target specification is also consistent.

The following "modularization" theorem provides general conditions for the generation of consistent refinements [11, 16].

$$\begin{array}{ccc} P & \xrightarrow{c} & S \\ \downarrow{\scriptstyle r} & & \downarrow{\scriptstyle r'} \\ P' & \xrightarrow{c'} & S' \end{array}$$

Theorem 1. *Let P, P', and S be first-order specifications, where $c : P \to S$ is a conservative extension and $r : P \to P'$ is a consistent refinement. If S' is the pushout with cocone morphisms $c' : P' \to S'$ and $r' : S \to S'$, then c' is a conservative extension and r' is a consistent refinement.*

Theorem 1 is typically applied when the goal is to refine a given specification S. A generic specification transformation based on the theorem performs the following steps (we present several instances below):

SpecTransformation(S:Spec):Morphism

1. analyze S
2. generate the refinement morphism $r : P \to P'$
3. generate a classification morphism $c : P \to S$ which shows how r applies to S
4. compute the pushout of $P' \xleftarrow{c} P \xrightarrow{r} S$ yielding cocone $P' \xrightarrow{c'} S' \xleftarrow{r'} S$
5. return r'.

The generated morphism $r' : S \to S'$ is the desired consistent refinement of S. The refinement r represents the core design decision and each transformation embodies its own class of design knowledge. The pushout extends its application to the whole specification. Theorem 1 provides the most general conditions known to us under which the generated refinement r' is consistent. A proof that r' is a consistent morphism from S and that it embodies an instance of the design knowledge codified by the transformation can be generated automatically at refinement-generation time [12].

3 Incrementally Constructing Sum/Variant/Inductive Types

A constructor for a type T is a function of type $c : A[T] \to T$ where $A[T]$ is a (possibly empty) product of auxiliary types and zero or more positive occurrences of T. A *base constructor* has a signature $c : A \to T$ with no occurrence of T in its domain. A constructor set is *well-founded* if it contains at least one base constructor. An inductive type is defined by a well-founded set of constructors (aka injections).

For example, the specification to the left in Figure 1 contains a well-founded set of constructors for the type of leaf-labeled binary trees, where `Empty` constructs the empty `BinTree`, `Leaf` constructs leaves labeled with natural numbers, and `Fork` builds a `BinTree` from a pair of (unlabeled) subtrees. There are many possible models of `BinTree`, but if we refine `BinTree` to `BinTree1` where `BinTree` is now defined as recursive variant type (i.e. named sum-type), then there is only one model (up to isomorphism). It also defines a function on `BinTree`s by means of pattern-based axioms that specify how `BinTreeDepth` behaves on each constructor. Overall, Figure 1 exemplifies the kind of refinement that we generate. This definition can be proved complete, consistent, and terminating using the induction rule for `BinTree`. The construction gives rise to an induction rule which reflects that, by construction, every element of the type is the valuation of a unique term built out of constructors, and conversely, that each term built out of constructors evaluates to a unique element

```
Bintree = spec
  type BinTree
  op Empty: BinTree
  op Leaf: Nat -> BinTree
  op Fork: BinTree*BinTree -> BinTree
  op BinTreeDepth: BinTree -> Nat
  ax BinTreeDepth Empty  = 0
  ax BinTreeDepth Leaf n = 1
  ax BinTreeDepth Fork(bt1,bt2)
        = max(BinTreeDepth(bt1),
              BinTreeDepth(bt2))
end-spec
```

$\xrightarrow{n}$

```
Bintree1 = spec
  type BinTree = | Empty | Leaf Nat
                 | Fork BinTree*BinTree
  op BinTreeDepth(bt:BinTree):Nat =
     case bt of
     | Empty  ->  0
     | Leaf n ->  1
     | Fork(bt1,bt2) -> max(BinTreeDepth(bt1),
                            BinTreeDepth(bt2))
end-spec
```

Fig. 1. Refinement to Inductive Bintree Specification

of the type. Under various conditions it is possible to allow axioms that, for example, identify two distinct terms over the constructors (e.g. to admit a commutative constructor). Our examples will not require this capability.

3.1 Incremental Accumulation of Constructors

```
Tspec = spec
  type T
  op c0:T
  op c1:Nat*T-> T
  op f:T->B
  ax f(c0) = b0
  ax f(c1(n,a1)) = b1
end-spec

Tspec1 = spec
  import Tspec
  op c2:T*T->T
  ax f(c2(a1,a2)) = b2
end-spec
```

The idea of incrementally defining an inductive type is simple. During a derivation, we introduce a new undefined type symbol and incrementally add constructors. We also introduce function symbols and incrementally add pattern-based axioms that specify how the function behaves on each constructor. In the end, the developer declares the constructor set complete and applies a transformation that defines the type as a sum/variant type and provides inductive definitions for the function symbols.

As an abstract example, `Tspec` introduces `T` as an undefined type that has two constructors `c0` and `c1`. `Tspec` also introduces `f` as an undefined function that is constrained by its type and by axioms that characterize its functionality by specifying how it behaves on the two constructors. `Tspec1` refines `Tspec` by (1) extending it with a new constructor `c2`, and (2) extending the characterization of `f` by showing how it behaves on the new constructor. `Tspec1` can be further extended in a similar manner.

3.2 CompleteSumType Transformation

At some point in a derivation, the developers decide that no more constructors are needed. The CompleteSumType transformation is then applied to generate a refinement in which `T` and its functions are given definitions. This is a strong refinement in the sense that it narrows down the possible interpretations of `T` and its functions from a possibly infinite set to a singleton – they are given canonical definitions (up to isomorphism).

We present the CompleteSumType transformation as an instance of the `SpecTransformation` transformation pattern in Section 2. We factor the transformation into two steps. The first, exemplified in Figure 2, analyzes the given specification `S` to abstract out a subspecification `Scons` that contains just the constructors over a given undefined type `T`. If the constructor set is well-founded, then it generates a refinement/morphism `r`:`Scons` $\rightarrow$ `Scons'`) where `Scons'` introduces a sum-type definition for `T` in place of the constructor axioms. It then generates a refinement of `S` by taking

```
Scons = spec
  type T
  op c0:T
  op c1:Nat*T-> T
  op c2:T*T->T
end-spec
```

$\xrightarrow{r}$

```
Scons' = spec
  type T = | c0 | c1 Nat*T | c2 T*T
end-spec
```

Fig. 2. Abstract Refinement Morphism

the pushout of `r` and the conservative extension `c`:`Scons` $\rightarrow$ `S`. It is straightforward to show that `r` is a consistent refinement, since it picks out the one model of `T` that is the least fixpoint of the well-founded constructor set. By Theorem 1, if *c* is conservative and `S`' is the pushout of `r` and `c`, then the generated refinement `r'`:S$\rightarrow$ `S'` is consistent.

The second step analyzes `S'` to abstract out a subspecification `Sfuns` that contains just the function symbols that have pattern-based axioms over the constructors in `Scons`. It then generates a refinement `n`:`Sfuns` $\rightarrow$ `Sfuns'` where `Sfuns'` introduces case-based definitions for each function in place of the inductive axioms, as exemplified in Figure 3. It then generates a refinement of `S'`

```
Sfuns = spec
  type T = | c0 | c1 Nat*T | c2 T*T
  op f:T->B
  axiom f(c0) = b0
  axiom f(c1(n,a1)) = b1
  axiom f(c2(a1,a2)) = b2
end-spec
```

$\xrightarrow{r}$

```
Sfuns' = spec
  type T = | c0 | c1 Nat*T | c2 T*T
  op f(a:T):B =
    case a of
     | c0        -> b0
     | c1(n,a1)  -> b1
     | c2(a1,a2) -> b2
end-spec
```

Fig. 3. Generated Refinement Morphism

by taking the pushout of `r` and the conservative extension `c`:`Sfuns` $\rightarrow$ `S'`. It is straightforward to show that `r` is a consistent refinement, using the induction rule that goes with the definition of a recursive sum-type. By Theorem 1, if `c` is conservative and `S''` is the pushout of `r` and `c`, then the generated refinement `r'`:S$\rightarrow$ `S''` is consistent. Note that specification `S` may have constraints on `f` beyond the pattern-based axioms, but the conservativeness of `c` requires that they imply no additional theorems.

3.3 Example: Specifying Reference Types

The need to specify and design programs that use references in Specware was a motivation for developing CompleteSumType. A key challenge is knowing the type of a reference. A polymorphic definition of a reference type doesn't allow retrieval of the underlying type. One solution is to maintain a ghost record of the current types at all memory locations, where the allowed types are those supported by the underlying architecture [7, 2, 15]. In a specification setting, it is necessary to reference user-introduced types that may not yet have a definition, so a more general mechanism is needed. Another solution is to directly tag all dynamically allocated objects with their type, although this is just an expensive variant of the first.

```
RefTypes = spec
  type State
  type Value
  type Ref
  op deref: State*Ref -> Value
end-spec
```

Our solution is to introduce an inductive type `Value` that represents all referenceable types in our application. It need not represent all possible types, just those that are used. It is desirable to be able to extend `Value` with new referenceable types (e.g. for a program family). `Ref` is the type of references, and a dereference function then determines in a given `State`, what the `Value` is of a given `Ref`.

During the refinement process, for each referenceable type `T` that is introduced, we introduce a new constructor for `T`. We also introduce testors (to decide if a `Value` represents a `T` element) and coercion/destructor functions (to invert a constructor).

```
Nat32Ref = spec
  import RefTypes
  type Nat32
  op c_Nat32: Nat32 -> Value
  op Nat32?(val:Value):Bool =
      (ex(pkt:Nat32) val = c_Nat32 pkt)
  op coerce_Nat32(val:Value | Nat32? val):
                 {pkt:Nat32 | val=c_Nat32 pkt}
end-spec

PacketRef = spec
  import RefTypes
   type Packet
   op c_Packet: Packet -> Value
   op pkt?(val:Value):Bool =
      (ex(pkt:Packet) val = c_Packet pkt)
   op coerce_Packet(val:Value | pkt? val):
                {pkt:Packet | val=c_Packet pkt}
   op data: Packet -> Int32
   op get_data(st:State, pktRef:Ref
              | pkt?(deref st pktRef)):Int32 =
      data(coerce_Packet(deref st pktRef))
end-spec
```

For example, specification `Nat32Ref`, introduces constructors for `Nat32` (eventually refining to unsigned 32-bit integers) and Packets (a user-defined type for use in communication software). The predicate `is_Nat32?` tests whether a `Value` represents a `Nat32`. The function `coerce_Nat32` coerces a `Value` back to a `Nat32` assuming that it represents a `Nat32`. `coerce_Nat32` would be implemented as a type cast in many programming languages. Analogous functions are introduced for the user-defined `Packet` type.
As a simple example, the function `get_data` takes in a reference to a `Packet` and returns the data value of the packet. In Section 4.3, we extend this development by allowing referenceable types that are also mutable.

3.4 Subtyping

One might want a family tree of sum-types and an appropriate notion of sum-type subtyping. The example in Figure 4 introduces `T` as an intended inductive type, and then introduces `T1` as an intended supertype `T1:>T`, and `T2` as another intended supertype `T2:>T`. We can then import `S1` and `S2` and transform as shown in Figure 5. In specification `S3'`, the function `f1` may be passed a `T` or `T1` element, and `f2` may be passed a `T` or `T2` element.

```
S = spec
  type T  % intended sum-type
  op c1: D1 -> T
  op c2: D2 -> T
  op f(a:T):A =
  ... pattern-based axioms
  over c1 and c2 ...
 end-spec
```

```
S1 = spec
  import S
  type T1 :> T
  op c3: D3 -> T1
  op f1(a:T1):A1 =
  ... pattern-based axioms
  over c1, c2, and c3 ...
 end-spec
```

```
S2 = spec
  import S
  type T2 :> T
  op c4: D4 -> T2
  op f2(a:T2):A2 =
  ... pattern-based axioms
  over c1, c2, and c4 ...
 end-spec
```

Fig. 4. Sum Subtype Development

```
S3 = spec
  import S1,S2
  ...
 end-spec
```

$\xrightarrow[\textsc{CompleteSumType}(S3,T1)\ \textsc{CompleteSumType}(S3,T2)]{\textsc{CompleteSumType}(S3,T)}$

```
S3' = spec
  type T  = |c1:D1 | c2:D2
  type T1 = |c1:D1 | c2:D2 | c3:D3 :> T
  type T2 = |c1:D1 | c2:D2 | c4:D4 :> T
  op f(a:T)   :A  = ... inductive def on T  ...
  op f1(a:T1):A1 = ... inductive def on T1 ...
  op f2(a:T2):A2 = ... inductive def on T2 ...
  ...
 end-spec
```

Fig. 5. Sum Refinement

4 Incrementally Constructing Product/Record Types

Suppose that our requirement modeling or design direction requires a type T but a priori we don't know its content. It may be natural to introduce constraints on T as needed during the derivation process in the form of additional observations of T. An *observer* of type T is a function $p : T \rightarrow A[T]$ where $A[T]$ is a a (possibly empty) product of auxiliary types and zero or more positive occurrences of T. An observer extracts information of type $A[T]$ from a T object.

For example, in a vehicle context, we might introduce a `State` type together with observations about the current time, and position of the vehicle, and a drive function that changes state; see specification `Vehicle` in Figure 6. Later we might add an observation of the vehicle's velocity; see specification `Vehicle1` in Figure 6. There are many possible models of `State`, but if we refine `Vehicle1` to `Vehicle2` where `State` is now defined as record type (i.e. named product), then there is only one model (up to isomorphism). The refinement in Figure 6 also defines a function that changes `State` by means of copattern-based axioms that specify `drive` in terms of observations of its output. This definition can be proved complete, consistent, and terminating using the coinduction rule that can be generated for `State`. Overall, Figure 6 exemplifies the kind of refinement that our CompleteProductType transformation generates.

A possibly-recursive record (named product) is defined in the form

```
type T= {p1 : A1[T], ..., pn: An[T]}
```

where `pi:T-> Ai[T]` for `1<=i<=n` is the complete set of observers of `T` (aka projections and fields). An element of `st:State` is written as a constant in the form

```
st = {time = 0, position = -1, velocity = 2}
```

and a functional update to a record is written

```
st << {time = 1, position = 1}
```

to denote a new record that differs from `st` only in the fields `time` and `position`:

```
{time = 1, position = 1, velocity = 2}
```
.

Streams provide a prototypical example of a recursive record type:

```
type Stream Nat = {hd:Nat, tl: Stream Nat}
```

Current thought is that possibly-recursive record types and especially the infinite objects in coinductive types are best understood in terms of their observers.

Deciding that the observers of `State` are complete, we can define `State` as a record of current observations, and then give a definition to `drive` by simply updating the input state to satisfy its copattern-based axioms. Completeness and consistency can be proved trivially by coinduction. The resulting specification can be readily translated to monadic or imperative form, when the occurrences of `T` are single-threaded. The construction of `T` gives rise to a coinduction rule which

```
Vehicle = spec
  type State
  op time     : State -> Nat
  op position : State -> Integer
  op drive:Integer->State->State
  ax time (drive newVel st) = time st + 1
  ax position (drive newVel st)
      = position st + newVel
end-spec

Vehicle1 = spec
  import Vehicle
  op velocity : State -> Integer
  ax velocity (drive newVel st) = newVel
end-spec
```

$\xrightarrow{n}$

```
Vehicle2 = spec
 type State = {time    : Nat,
               position: Integer,
               velocity: Integer}

 op drive(newVel:Integer)(st:State):State
      = st << {    time = st.time + 1,
               position = st.position + newVel,
               velocity = newVel}
      end-spec
```

Fig. 6. Refinement to Record-based Coinductive Vehicle Specification

reflects that, by construction, every element of the type is uniquely identified by its observed values. Intuitively, if we cannot distinguish two elements of `T` through any sequence of observations, then the elements are equal. With some care it is possible to allow axioms that, for example, require a relationship between observers (e.g. that $p_1(t) \leq p_2(t)$ for all $t \in \mathtt{T}$). Our examples will not require this capability.

4.1 CompleteProductType Transformation

The idea of incrementally defining a coinductive type `T` is simple. During a derivation, we introduce a new type symbol and incrementally add observers. We also introduce function symbols and incrementally add copattern-based axioms on them. At some point in the derivation, the developers decide that no more observers are needed on type `T`. The CompleteProductType transformation is then applied to generate a refinement in which `T` and its functions are given definitions. This is a strong refinement in the sense that it narrows down the possible interpretations of `T` and its functions from a possibly infinite set to a singleton – they are given canonical definitions (up to isomorphism).

We present the CompleteProductType transformation as an instance of the `SpecTransformation` transformation pattern in Section 2. As before, we factor the transformation into two steps. The first, shown in Figure 7, analyzes the given specification `S` to abstract out a subspecification `Sobservers` that contains just the observers over a given undefined type `T`. It then generates a refinement/morphism `r`:`Sobservers` → `Sobservers'` where `Sobservers'` introduces a record-type definition for `T`, as exemplified in Figure 7. It then generates a refinement of `S` by taking the pushout

```
Sobservers = spec                         Sobservers' = spec
  type T                         r          type T = {p0:1, p1:Nat, p2:T}
  op p0:T -> 1                 ----->     end-spec
  op p1:T -> Nat
  op p2:T -> T
end-spec
```

Fig. 7. Abstract Refinement Morphism on Type T

of `r` and the extension `c`:`Sobservers` → `S`. It is straightforward to show that `r` is a consistent refinement, since it picks out the one model of `T` that is the greatest fixpoint of the recursive record type. By Theorem 1, if *c* is conservative and `S`' is the pushout of `r` and `c`, then the generated refinement `r'`:S→ `S'` is consistent.

The second step analyzes `S'` to abstract out a subspecification `Sfuns` that contains just the function symbols that have copattern-based axioms over the observers in `Sobservers`. It then generates a refinement `r`:`Sfuns` → `Sfuns'` where `Sfuns'` introduces record update definitions for each function, as exemplified in Figure 8. Since the function definition is co-recursive, and producing a recursive record-type may not terminate, its translation to a programming language must be handled with care. The transformation then generates a refinement of `S'` by taking the pushout of `r` and the

```
Sfuns = spec                                Sfuns' = spec
  type T = {p0:1, p1:Nat, p2:T}               type T = {p0:1, p1:Nat, p2:T}
  ax p0(f(a)) = e0(a)               r         op f(a:A):T = {p0 = e0(a), p1 = e1(a), p2 = e2(a)}
  ax p1(f(a)) = e1(a)             ---->     end-spec
  ax p2(f(a)) = e2(a)
end-spec
```

Fig. 8. Abstract Refinement Morphism for a Coinductively Defined Function

extension `c`:`Sfuns` → `S'`. It is straightforward to show that `r` is a consistent refinement, using the coinduction rule that goes with the definition of a recursive record type. By Theorem 1, if `c` is conservative and `S'` denotes the pushout of `r` and `c`, then the generated refinement `n'`:S→ `S'` is consistent.

4.2 Example: Packets

Communication streams provide a source of examples for incremental construction, which we illustrate by developing network-layer and transport-layer packet structures.

```
BasicPacket = spec
   type Data
   type Packet
   op data: Packet -> Data
end-spec
```

`BasicPacket` introduces a `Packet` type and one observer `data` of the content of a `Packet` which has some unspecified type `Data`.

```
TransportPacket = spec
   import BasicPacket
   type Port = Nat16
   op srcPort,dstPort: Packet -> Port
   op SeqNum : Packet -> Nat32
end-spec
```

`TransportPacket` extends `BasicPacket` with observers of a packet's source port `srcPort`, its destination port `dstPort`, and a sequence number `SeqNum`.

```
NetworkPacket = spec
   import BasicPacket
   type NetAddr = Nat32
   op srcAddr,dstAddr: Packet -> NetAddr
   op pktLen : Packet -> Nat16
end-spec
```

`NetworkPacket` extends `BasicPacket` with observers of a packet's source address `srcAddr`, its destination address `dstAddr`, and packet length `pktLen`. The types `Nat16` and `Nat32` are subtypes of `Nat` restricted to $[0..2^{16})$ and $[0..2^{32})$ respectively.

```
FlatNetTransPacket = spec
  import NetworkPacket,
         TransportPacket
end-spec
```

$\xrightarrow{\text{FCT}}$

```
FlatNetTransPacket1 = spec
  import BasicPacket
  type Port
  type NetAddr = Nat32
  type Packet =
       {srcAddr, dstAddr: NetAddr,
        pktLen : Nat16,
        srcPort, dstPort: Port,
        SeqNum : Nat32, data: Data}
end-spec
```

`FlatNetworkTransportPacket` incorporates the observers of `BasicPacket`, `NetworkPacket`, and `TransportPacket`. The refinement `m` generated by CompleteProductType creates the record-type definition for `Packet`.

A variation on the above formulation of `Packet` would distinguish header information (i.e. metadata) from payload content (i.e. data).

```
BasicPacket = spec
   type Data
   type Packet
   op data: Packet -> Data
   type Metadata
   op metadata: Packet -> Metadata
end-spec
```

Extending this specification with various observers of `Metadata` would give rise to the familiar header structures of the TCP/IP stack, by applying CompleteProductType to both `Packet` and `Metadata`.

4.3 Example: Mutable Types

Suppose that we wish to treat `Packet`s as dynamically allocated mutable objects. This example combines the development of both inductive types (references from Section 3.3) and coinductive types (`Packet` from Section 4.2). Continuing the example from Section 4.2, specification

```
MutableBasicPacket = spec
  import RefTypes, BasicPacket
  op c_Packet: Packet -> Value
  op pkt?(val:Value):Bool = (ex(pkt:Packet) val = c_Packet pkt)
  op coerce_Packet(val:Value | pkt? val): {pkt:Packet | val=c_Packet pkt}

  op get_data(st:State)(pktref:Ref | pkt?(deref st pktref)): Data
             = data(coerce_Packet(deref st pktref))
  op set_data(st:State)(pktref:Ref | pkt? (deref st pktref))(d:Data):
             {(pktRef,st'):Ref*State | get_data st' pktRef = d }
  op new_Packet(d:Data)(st:State): {(pktRef,st'):Ref*State |  pkt? (deref st' pktRef)
                                   && get_data st' pktRef = d}
end-spec
```

Fig. 9. MutableBasicPacket

`MutableBasicPacket` introduces a coinductive type `Packet` together with its `data` observer. Continuing the example in Section 3.3, we treat `Packet` as a referenceable type by introducing a constructor `c_Packet` of inductive type `Value`. We also introduce a defined observer `get_data` of `State` which observes the `data` of the `Packet` referenced by the argument `pktref`. We also specify a `State` transformer `set_data` that has the effect of changing the `data` observation of the `Packet` referenced by the argument `pktref`. Finally, we introduce a constructor of `Packet` that returns a reference. Effectively, `MutablePackets` provides a class-like specification, with a constructor, observers for getter methods, and transformers serving as setters and other methods. Translation to a suitable object-oriented language such as Java would be straightforward.

As in the previous section, we can extend `Packet` structure with Network structure by adding observers `srcAddr`, `dstAddr`,and `pktLen`, and their corresponding getters and setters. We also define a constructor for `MutableNetworkPacket`, which effectively becomes a subclass of `MutableBasicPacket`. See Figure 10.

4.4 Example: Mutable Heaps for a Garbage Collector

One motivation for the development of the CompleteProduct transformation was the derivation of a family tree of garbage collectors [10, 13] that we carried out using the Specware system [5].

cotype Graph	
Observers	**Functions**
nodes	add/delete node
nodeValue	set node value
arcs	add/delete arc
source (of an arc)	
target (of an arc)	setTarget (ptr swing)

In this context a model of memory starts with a directed graph type with observers for the nodes and arcs and associated observers of the content or value of a node and the source and target of each arc. Various functions for adding/deleting nodes and arcs, setting the value of a node, and setting the target of an arc are characterized using copattern-based axioms.

cotype Heap (extending Graph)	
Observers	**Functions**
roots	add/delete root
supply	add/delete supply node
	allocate node

The Graph specification is general-purpose and reusable, but a collector also needs to extend it to model the runtime heap, with additional observer for roots to specify the registers and stack sources of pointers, and the supply of nodes that can be dynamically allocated.

```
MutableNetworkPacket = spec
  import MutableBasicPacket
  type NetAddr = Nat32
  op srcAddr, dstAddr: Packet -> NetAddr
  op pktLen : Packet -> Nat16
  op get_srcAddr(st:State)(pktref:Ref | pkt?(deref st pktref)):Nat16
           = srcAddr (coerce_Packet (deref st pktref))
  op set_srcAddr(st:State)(pktref:Ref | pkt?(deref st pktref))(saddr:Nat16):
                          {(pktRef,st'):Ref*State |  get_data st' pktRef = get_data st pktRef
                          && get_srcAddr st' pktRef = saddr }
  ... similar definitions for get/set_dstAddr ...
  op get_pktLen(st:State)(pktref:Ref | pkt? (deref st pktref)):Nat16
            = pktLen (coerce_Packet (deref st pktref))
  op set_pktLen(st:State)(pktref:Ref | pkt? (deref st pktref))( len:Nat16):
                         {(pktRef,st'):Ref*State | get_data st' pktRef = get_data st pktRef
                          && get_pktLen st' pktRef = len}
  op new_NetworkPacket(st:State)(d:Data)(saddr:NetAddr)(daddr:NetAddr)(pktlen:Nat16):
                      {(pktRef,st'):Ref*State |  pkt? (deref st' pktRef)
                        && get_data st' pktRef = d
                        && get_srcAddr st' pktRef = saddr
                        && get_dstAddr st' pktRef = daddr
                        && get_pktLen  st' pktRef = pktlen}
end-spec
```

Fig. 10. MutableNetworkPacket

```
S = spec
  type T
  op p1: T -> D1
  op p2: T -> D2
  op g(a:A):T =
  ... copattern-based axioms
  over p1 and p2 ...
 end-spec
```

```
S1 = spec
  import S
  type T1 <: T
  op p3: T1 -> D3
  op g1(a:A1):T1 =
  ... copattern-based axioms
  over p1, p2, and p3 ...
 end-spec
```

```
S2 = spec
  import S
  type T2 <: T
  op p4: T2 -> D4
  op g2(a:A2):T2 =
  ... copattern-based axioms
  over p1, p2, and p4 ...
 end-spec
```

Fig. 11. Record Subtype Development

CollectionHeap (extending Heap)	
Observers	**Functions**
black	insert/delete black
	mark, sweep

Finally, we add observers that are needed by the particular collection algorithm. For example, a mark-and-sweep algorithm requires an observer of the mark bit (called black by tradition dating to Dijkstra) per node, and associated functions.

4.5 Subtyping

One might want a family tree of record-types and an appropriate notion of record-type subtyping, as illustrated in Figure 11. We can import `S1` and `S2` and transform as shown in Figure 12. In specification `S3'`, the function `g` may be passed a `T`, `T1`, or `T2` element, but `g1` may only be passed a `T1` element and `g2` may only be passed a `T2` element. This transformation naturally leads to the development of object class hierarchies in an object-oriented language.

```
S3 = spec
  import S1,S2
  ...
 end-spec
```

$$\xrightarrow[\substack{\textsc{CompleteProductType}(\text{S3,T1})\\ \textsc{CompleteProductType}(\text{S3,T2})}]{\textsc{CompleteProductType}(\text{S3,T})}$$

```
S3' = spec
  type T  = {p1:D1, p2:D2}
  type T1 = {p1:D1, p2:D2, p3:D3} <: T
  type T2 = {p1:D1, p2:D2, o4:D4} <: T
  op g(a:A)  :T  = ... coinductive def over T  ...
  op g1(a:A1):T1 = ... coinductive def over T1 ...
  op g2(a:A2):T2 = ... coinductive def over T2 ...
  ...
 end-spec
```

Fig. 12. Subtype Record Refinement

5 Implementation

An implementation of CompleteProductType must gather the observers on a given type symbol `T`. Observers are functions of a particular type $T \to A$ for some A that is not `T`. Of these, there are three subclasses of observers that can arise in a derivation: (1) undefined observers, (2) defined observers that are eagerly maintained (e.g. by a Finite Differencing transformation [9, 8]), (3) defined observers that are computed when needed. Only the observers in classes 1 and 2 are gathered for inclusion in the state definition. Class 3 is excluded for efficiency reasons only, under the presumption that it is called infrequently. If it is called frequently, then it may be more efficient to maintain a state variable for its value under all transformers (in which case it falls under class 2).

The transformations have been implemented in the Specware system [5] and have been used extensively in the automated synthesis of concurrent garbage collection algorithms [10, 13] and families of protocol-processing codes for distributed vehicle control [6].

6 Related Work

A goal of programming languages and programming paradigms is to ease the cost of software evolution, making it easier to add new features and to adapt to changing requirements. The *expression problem* arises from the desire to define a type and its functions incrementally by cases while preserving static type checking and avoiding recompilation [3, 17]. It was observed that (1) in functional languages it is easy to add new functions but not to add cases to a type, and (2) in object-oriented languages it is easy to add cases to a type, but not to add new functions. The refinement setting provides a simple natural approach for developers (1) to incrementally add cases/constructors to an inductive type, and (2) to extend functions that are defined inductively over the constructors, and to add new functions. Dually, developers can incrementally add new observers to a coinductive type, and extend functions/transformers that are defined coinductively with respect to the observers.

The literature on (co)algebra has long noted the duality of defining functions that input an inductive type by patterns versus defining functions that produce a coinductive type by copatterns[4]. Recent work by Abel et al.[1] laid the foundation for integrating this duality into Haskell and other programming languages by formalizing patterns and copatterns, and supporting pattern-based definitions for inductive functions, and copattern-based definitions for functions returning a coinductive type.

7 Concluding Remarks

We hope that the view expressed herein offers a richer understanding of programs and program development in general. Algebraic/inductive datatypes and functions are useful for specifying immutable finite data structures. They naturally support a functional programming style. Coalgebraic/coinductive datatypes and functions are useful for specifying mutable data structures, non-well-founded data structures, as well as dynamical systems that are possibly nonterminating and concurrent. They naturally support imperative, object-oriented, and multi-threaded programming styles. Together they provide a natural foundation for a mixed use of functional, imperative, object-oriented and concurrent programming. Embedding these dual concepts in a refinement setting provides flexibility in the face of pressures to vary software, either to produce the products of a product line (via alternative product requirements), or to respond to evolutionary changes to requirements.

Acknowledgements: Thanks to Christoph Kreitz, Peter Pepper, and Florian Rabe for helpful discussions and comments on the text.

References

1. Abel, A., Pientka, B., Thibodeau, D., and Setzer, A. Copatterns: Programming infinite structures by observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2013), POPL '13, pp. 27–38.
2. Appel, A. W., Dockins, R., Hobor, A., and Beringer, L. *Program Logics for Certified Compilers.* Cambridge University Press, 2014.
3. Cook, W. Object-oriented programming versus abstract data types. In *Proc. REX workshop on Foundations of Object-Oriented Languages* (1990), Springer-Verlag LNCS 489.
4. Jacobs, B., and Rutten, J. A tutorial on (co)algebras and (co)induction. *Bulletin of the EATCS 62* (1996).
5. Kestrel Institute. *Specware System and documentation*, 2003. http://www.specware.org/.
6. Kreitz, C., and Smith, D. R. Synthesis of Network Protocols: Final Report. Tech. rep., Kestrel Institute, 2016. http://www.kestrel.edu/home/people/smith/pub/HACMS-Final-Report.pdf.
7. Leroy, X., and Blazy, S. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning 41*, 1 (2008), 1–31.
8. Liu, Y. *Systematic Program Design: From Clarity to Efficiency.* Cambridge University Press, 2013.
9. Paige, R., and Koenig, S. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst. 4*, 3 (July 1982), 402–454.
10. Pavlovic, D., Pepper, P., and Smith, D. R. Formal derivation of concurrent garbage collectors. In *Proceedings of 10th International Conference on Mathematics of Program Construction (MPC 2010)* (2010), C. Bolduc, J. Desharnais, and B. Ktari, Eds., vol. 6120, Springer Verlag, pp. 353–376. extended version in http://arxiv.org/abs/1006.4342.
11. Smith, D. R. Another proof of the modularization theorem. Tech. rep., Kestrel Institute, February 1993. http://www.kestrel.edu/home/people/smith/pub/modularization.pdf.
12. Smith, D. R. Generating programs plus proofs by refinement. In *Verified Software: Theories, Tools, Experiments* (2008), B. Meyer and J. Woodcock, Eds., Springer-Verlag LNCS 4171, pp. 182–188.
13. Smith, D. R., Westbrook, E., and Westfold, S. J. Deriving Concurrent Garbage Collectors: Final Report. Tech. rep., Kestrel Institute, 2015. http://www.kestrel.edu/home/people/smith/pub/CRASH-Final-Report.pdf.
14. Srinivas, Y. V., and Jüllig, R. Specware: Formal support for composing software. In *Proceedings of the Conference on Mathematics of Program Construction*, B. Moeller, Ed. LNCS 947, Springer-Verlag, Berlin, 1995, pp. 399–422.
15. Tuch, H. Formal verification of C systems code: Structured types, separation logic and theorem proving. *Journal of Automated Reasoning 42*, 2 (2009), 125–187.
16. Veloso, P. A., and Maibaum, T. On the modularization theorem for logical specification. *Information Processing Letters 53*, 5 (1995), 287–293.
17. Wadler, P. The expression problem. Tech. rep., Bell Labs, Murray Hill, NJ, 1998. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.