Testing versus proving in climate impact research

Cezar Ionescu[1] and Patrik Jansson[2]

1 **Potsdam Institute for Climate Impact Research**
Telegrafenberg A31, 14473 Potsdam, Germany
ionescu@pik-potsdam.de
2 **CSE Department, Chalmers University of Technology**
SE - 412 96 Göteborg, Sweden
patrikj@chalmers.se

Abstract

Higher-order properties arise naturally in some areas of climate impact research. For example, "vulnerability measures", crucial in assessing the vulnerability to climate change of various regions and entities, must fulfill certain conditions which are best expressed by quantification over all increasing functions of an appropriate type. This kind of property is notoriously difficult to test. However, for the measures used in practice, it is quite easy to encode the property as a dependent type and prove it correct. Moreover, in scientific programming, one is often interested in correctness "up to implication": the program would work as expected, say, if one would use real numbers instead of floating-point values. Such counterfactuals are impossible to test, but again, they can be easily encoded as types and proven. We show examples of such situations (encoded in Agda), encountered in actual vulnerability assessments.

1998 ACM Subject Classification D.1.1 Applicative (Functional) Programming, D.1.6 Logic Programming, D.2.4 Software/Program Verification, D.2.5 Testing and Debugging

Keywords and phrases dependently-typed programming, domain-specific languages, climate impact research, formalization

Digital Object Identifier 10.4230/LIPIcs.TYPES.2011.41

1 Introduction

Climate impact research is not the same as climate research: it does not deal, for example, with building the detailed simulations of the climate system that run on massively parallel machines of incredible, yet always insufficient computational power. Rather, climate *impact* research attempts to analyze the broad, first-order effects of various policies meant to mitigate or alleviate the problems caused by human-induced climate change. The Potsdam Institute for Climate Impact Research (the acronym *PIK* comes from the more compact German version: *Klimafolgenforschung*) has on its web page the following introduction:

> At PIK researchers in the natural and social sciences work together to study global change and its impacts on ecological, economic and social systems. They examine the Earth system's capacity for withstanding human interventions and devise strategies for a sustainable development of humankind and nature.
> PIK research projects are interdisciplinary and undertaken by scientists from the following Research Domains: Earth System Analysis, Climate Impacts and Vulnerabilities, Sustainable Solutions and Transdisciplinary Concepts and Methods.
> Through data analysis, computer simulations and models, PIK provides decision makers with sound information and tools for sustainable development. In addition to publishing results in scientific journals the Institute gives advice to national and regional authorities and, increasingly, to global organisations such as the World Bank.
>
> (from http://www.pik-potsdam.de/institute)

18th International Workshop on Types for Proofs and Programs (TYPES 2011).
Editors: Nils Anders Danielsson, Bengt Nordström; pp. 41–54

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The important point here is the following: many complex systems are studied together by scientists from many different disciplines. In this kind of enterprise, the concepts that tend to be most used across disciplines have a high intuitive content, which ensures that they are quickly grasped by all the different parties ("vulnerability" will be our running example, but consider also "stability", "resilience", "global change", "sustainable growth", "green path", and so on). The danger is that each party will grasp it in a different way, hence the importance of definitions. In general, the more formal the definition, the less the risk it will be misunderstood (though the chance of being understood might also decrease), and here is where a first connection to logic and computer science appears.

Additionally, such "fulcrum" concepts that leverage our everyday intuitions and help structure the interdisciplinary discourse also provide natural candidates for assessments, for measurement and comparison, which then, in turn, can be used as the basis for "giving advice to national and regional authorities". Many of these assessments are computer-based, and subject to the usual concerns of reuse, genericity, efficiency and correctness (especially important, one would think, when giving advice "to global organizations such as the World Bank").

This is the computer scientists' playground, and the game plan is: formalize the concepts involved in order to be able to write specifications against which to assess program correctness. Do it generically, in order to unify and reuse as much as possible of the existing code. Since the subject is largely mathematical, use a high-level language with an expressive type system, in order to minimize the distance from specification to implementation. Hopefully, the end-result will be a domain-specific language, which will simplify writing the particular sort of programs we started with, while at the same time making their correctness easier to assess.

This paper presents some of the results we obtained while playing this game within the field of (computer-assisted) *vulnerability assessment*. The next section is a whirlwind tour of definitions of vulnerability and the resulting (simplified) Haskell formalization. We then take up the question of correctness: we want to ensure that key conditions are met by an implementation. The first idea, presented in Section 3, is in tune with current software engineering best practices: apply automatic property-based testing (for example, using QuickCheck [8]). It turns out that writing good tests is somewhere between hard and impossible, but proving on paper that the conditions hold is really easy. Therefore, we re-implemented parts of the system in a dependently-typed programming language (Agda[1], [21, 26]) and found that expressing the conditions as types was at least as easy as thinking up good tests, and that convincing the type checker that the conditions were met was at least as easy as implementing those tests. Moreover, things that were impossible before become not even hard. This is presented in Section 4, which raises questions such as: if proving things is so easy, why does it get such a bad reputation? We have an opinion about this, and you can read it in the conclusions.

2 Vulnerability

In the past decade, the concept of "vulnerability" has played an important role in fields such as climate change, food security and natural hazard studies. Vulnerability studies have often been successful in alerting policymakers to precarious situations. The importance of the

[1] The choice of Agda over, say, Coq, was motivated partly by similarity with Haskell (since we could translate our Haskell code-base), partly by aesthetic considerations and by ease of use. Perhaps the largest role was played by the fact that PIK has quite close ties to Chalmers, where Agda was developed.

concept in the particular field of climate change is described, for example, as follows [13]:

> ... Studies based primarily on the output of climate models tend to be characterized by results with a high degree of uncertainty and large ranges, making it difficult to estimate levels of risk. In addition, the complexity of the climate, ecological, social and economic systems that researchers are modeling means that the validity of scenario results will inevitably be subject to ongoing criticism. ... Such criticisms should not be interpreted as questioning the value of scenarios; indeed, there is no other tool for projecting future conditions. What they do, however, is emphasize the need for a strong foundation upon which scenarios can be applied, a foundation that provides a basis for managing risk despite uncertainties associated with future climate changes. This foundation lies in the concept of vulnerability.

No doubt, vulnerability is one of the "fulcrum" concepts mentioned in the introduction and, alerted to the importance of definitions in an interdisciplinary context, we expect this one to be very well defined. Unfortunately, this is only the case if by "well defined" we mean "defined many times". Figure 1 contains a sample of vulnerability "definitions" found in the literature:

- [16]: Vulnerability is defined as the extent to which a natural or social system is susceptible to sustaining damage from climate change. Vulnerability is a function of the sensitivity of a system to changes in climate (the degree to which a system will respond to a given change in climate, including beneficial and harmful effects), adaptive capacity (the degree to which adjustments in practices, processes, or structures can moderate or offset the potential for damage or take advantage of opportunities created by a given change in climate), and the degree of exposure of the system to climatic hazards.
- [28]: The conditions determined by physical, social, economic, and environmental factors or processes, which increase the susceptibility of a community to the impact of hazards.
- [7] Vulnerability, therefore, is a human-induced situation that results from public policy and resource availability/distribution, and it is the root cause of many disaster impacts. Indeed, research demonstrates that marginalized groups invariably suffer most in disasters. Higher levels of vulnerability are correlated with higher levels of poverty, with the politically disenfranchised, and with those excluded from the mainstream of society.
- [6] Vulnerability (in contrast to poverty which is a measure of current status) should involve a predictive quality: it is supposedly a way of conceptualizing what may happen to an identifiable population under conditions of particular risk and hazards. Is the complex set of characteristics that include a person's: initial well-being (health, morale, etc.); self-protection (asset pattern, income, qualifications, etc.); social protection (hazard preparedness by society, building codes, shelters, etc.); social and political networks and institutions (social capital, institutional environment, etc.).
- [9] Vulnerability (V) = Hazard Coping,
  with Hazard = H (Probability of the hazard or process; shock value; predictability; prevalence; intensity/strength);
  and Coping = C (Perception of risk and potential of an activity; possibilities for trade; private trade, open trade).

Figure 1 A sample of vulnerability definitions from several different papers.

There are many, many more such definitions, a large percentage of which wouldn't pass Pascal's requirement of "application of a name to things which are clearly designated by

terms perfectly known" [25]; the curious reader is referred to Thywissen's summary of some thirty-odd definitions [27].

There is a corresponding diversity in the way in which vulnerability is *measured*. Examining the technical details of computer-assisted vulnerability assessments is tedious, but has a clear advantage over reading definitions such as the above: one can unambiguously determine what is being measured.

Virtually all vulnerability assessments have the following structure. First, one tries to estimate the evolution of various parameters of interest, for example, the average temperature in a given region, the gross domestic product of a country, the sea-level of some coastal area, but also less immediately relevant values, such as literacy rate or number of telephone lines in a region [17]. Sometimes, the result of this forecasting analysis is a list of values, one element for each time period (week, month or year) of the time horizon (typically measured in decades). Most times, the result will consist of several such trajectories, perhaps with some additional information about their likelihood. Thus, one can have lists of possible trajectories, or a probability distribution over trajectories, or a fuzzy set of trajectories, etc.

Next, each trajectory is examined in order to determine the harm that befalls the region or population under consideration: damages, negative impacts, losses caused by the factors of interest (for example, human-induced climate change). Harm is represented in many ways, but it is always assumed that the resulting values can be at least partially compared, i.e., that they are members of a preordered set.[2]

Depending on how the forecast of the parameters was achieved, we have so far a list of harm values, or a probability distribution over harm values, or a fuzzy set, etc. Now comes the final step: aggregating all these harm values, obtaining the final vulnerability assessment. This is usually done either by taking some representative value, for example the maximal or the likeliest harm, or by an integral measure of the possibilities (such as their sum or average). The final value does not need to lie in the same set as the harm values, but vulnerability values also need to form at least a preorder: the purpose of the assessment is often to compare the relative vulnerabilities of regions, or of the same region under different scenarios.

In Haskell, these explanations can be expressed more concisely and precisely:

```
data State = ...                          -- an appropriate type for the values of the
                                          -- parameters of interest
type Trajectory = [State]                 -- a trajectory is a list of states
type Possible = ...                       -- a functor which represents the structure
                                          -- of possible trajectories, e.g. List
data V = ...                              -- datatype of harm values
instance Preorder V where ...             -- harm values must be preordered
data W = ...                              -- datatype for vulnerability values
instance Preorder W where ...             -- vulnerability values must be preordered
vulnerability :: Possible Trajectory →    -- possible trajectories
                 (Trajectory → V) →       -- harm evaluation
                 (Possible V → W) →       -- aggregation of harm values
                 W                        -- type of final result
vulnerability possible harm measure = measure (fmap harm possible)
```

[2] The reason for not requiring anti-symmetry is that harm values are often compared via cost functions.

Possible trajectories are collected together in a functorial structure. Besides the fact that all our examples (probability distributions, fuzzy sets, lists) are functors, this makes sense because of the need to apply the harm evaluation function to each trajectory. Otherwise, the code follows literally the description above.

Most of the work in a vulnerability assessment is put in computing the structure of possible trajectories. To do this, existing models are used (and reused), which are usually written by specialists in the relevant disciplines: economists, climate scientists, geographers, social scientists, etc. The models are then combined by the team that does the vulnerability assessment. Sometimes, these models have different types: a climate model might yield a deterministic trajectory of the average global temperature, while a demographic model might offer only a list of possible evolutions of the population, and an economic model a probability distribution over possible future values of the gross domestic product. Accordingly, most of the work we have done was in extracting the general structure of these models and of the means of combining them, in order to simplify the task of the vulnerability assessment in its most difficult part. The result was a domain-specific language for describing and combining *monadic dynamical systems*, described extensively by Ionescu [11] and concisely by Lincke et al. [14].

Here, however, we concentrate on the computationally less intensive part: the interplay between the evaluation of harm and the measurement of vulnerability. There is very little one can say to better describe the possible candidates for these functions: one cannot claim, for example, that only certain preordered sets are suitable and exclude others. But there is a condition which virtually everybody agrees on: if the harm evaluations along all trajectories in a structure are increased, then the vulnerability measure should also increase. This kind of monotonicity can be taken as the defining condition for a vulnerability measure:

▶ **Definition 1.** *Let V and W be two preorders, and F a functor. A function $m : F\ V \to W$ is called a* vulnerability measure *if, for any increasing function $i : V \to V$ (that is, $v \leqslant i\ v$ for all $v : V$), and any $x : F\ V$ we have $m\ x \leqslant m\ (F\ i\ x)$.*

If we use the order $x \sqsubseteq_m y = m\ x \sqsubseteq m\ y$ on $F\ V$ we can say that m is a vulnerability measure if "$(F\ i)$ is increasing when i is increasing". We will use this formulation in Section 4. No matter how good the models used to forecast the possible trajectories are, no matter how well combined, if a vulnerability assessment uses a function which is not a vulnerability measure in order to aggregate the harm values, then it must be regarded as flawed.

Are there any vulnerability assessments which fail in this respect? Unfortunately, yes. The "likeliest harm value" we mentioned above does not fulfill this condition, and neither do other "democratic" methods (the most frequent result of harm values, for instance). There is, therefore, scope for error, and so we come to the idea of *testing*, for a given implementation, that the vulnerability measure condition holds.

3 Testing vulnerability measures

To test a candidate vulnerability measure $m : F\ V \to W$ we first turn to the question of the functoriality of the structure of type $F\ V$ that collects the harm values. How do we know that the implementation of the mapping function preserves identities and compositions? The Haskell type system does not detect the problem with

```
mapTry :: (a → b) → [a] → [b]
mapTry f []       = []
mapTry f (a : as) = mapTry f as
```

The problem is that $mapTry\ id\ = const\ [\,] \neq id$, so the first functor law fails (but the second functor law holds). As an aside, *mapTry* is the version suggested by Agda's automatic theorem prover / type inhabitant searcher, called Agsy [15]. (To Agsy's defence should be said that it only aims at, and succeeds in, finding *some* value of the correct type.)

If we want to test if polymorphic properties like the functor laws hold for a polymorphic function like *mapTry*, we need to pick some monomorphic type to test them on. It is not in general enough to pick a trivial type like () or a small type like *Bool*, but most often it is enough to test with the type of natural numbers. For the functor laws the results of Bernardy et al. [2] allow us to reduce testing the polymorphic map function to just one type (and in fact, just we can even fix the function argument f), but there is still the question of coverage:

```
map :: (a → b) → [a] → [b]
map f []       = []
map f (a : as) = if length as ⩾ bigNumber
                   then map f as
                   else f a : map f as
```

Granted, this is a malicious example, but the problem remains, especially in the case of functors that require more complex implementations (such as the simple probability functor). Still, let us accept for now that the implementation of the mapping function is likely to be used in many programs and therefore verified in so many different cases that we can take it to be correct.

For concreteness, let us fix the functor to be the non-empty list functor given by

```
data List a = Wrap a | Cons a (List a)
  deriving (Ord, Eq, Show)

fold :: (a → b) → (a → b → b) → List a → b
fold w c (Wrap a)    = w a
fold w c (Cons a as) = c a (fold w c as)

instance Functor List where
  fmap f = fold (Wrap ∘ f) (λ a bs → Cons (f a) bs)
```

A typical type for harm values is a tuple: pairs of floating-point numbers representing (monetary) damages and natural numbers representing lost lives. The least controversial way of comparing such values is given by the dominance relation:

```
instance POrd a where
  leq :: a → a → Bool

instance (POrd a, POrd b) ⇒ POrd (a, b) where
  (a1, b1) `leq` (a2, b2) = a1 `leq` a2 ∧ b1 `leq` b2
```

We defined a new type class for preorders, similar to the *Ord* class provided by Haskell. Instances of the Haskell *Ord* class are required to be total orders, while instances of *POrd* should be preorders. Neither of these requirements can be expressed in Haskell, so there is no automatic check that instances really satisfy them. Anyway, let us grant that the preorder properties also do not need to be tested here (either because they are tested elsewhere, or because the implementation can be trivially seen to be correct).

The biggest problem that we encounter in testing vulnerability measures is its higher-order nature, namely the quantification over all possible increasing functions. In QuickCheck notation, one might write

```
testMonotonicity m i x = increasing i ==> m x `leq` m (fmap i x)
```

This naive translation of the requirement would check that i is an increasing function, and then check that v assigns an increased measure to the increased x. Even assuming the unlikely case in which the property of being increasing is decidable (this only works for functions with finite domain – not the case in our example), we still have the problem that arbitrarily generated functions are unlikely to be increasing, and QuickCheck will stop with an inconclusive result once it reaches the maximum number of attempts for which it is configured.

Thus, we need to use a custom generator which guarantees that the functions it generates are increasing:

```
testMonotonicity m genInc x = forAll genInc (λ i →
                                m x `leq` m (fmap i x))
```

The problem of coverage will still stay with us, but at least we can ensure that we reach the test of m. For the concrete example we have taken, we can, for example, implement a custom generator by:

```
genInc :: Gen ((Float, Int) → (Float, Int))
genInc = do dx ← choose (0, 10)
            dn ← choose (0, 10)
            return (λ (x, n) → (x + dx, n + dn))
```

and, in fact, we have done so [11]. Unfortunately, this can cause an error: large integers can overflow and result in large negative integers. To do a proper job, the generator has to examine its arguments, and make sure that the returned values really fulfill the desired condition.

Even with the best generator, we still have a problem. Consider a measure which just sums up the elements of the list of potential results:

```
sumList :: List (Float, Int) → (Float, Int)
sumList = fold id f
  where f (x, n) (x′, n′) = (x + x′, n + n′)
```

This should be a vulnerability measure: increasing the values in a list increases their sum. However, testing it can again fail if the integral part overflows, or if summing up the floating point leads to round-off errors. This means that we need to control also the generation of the arguments, not just the generation of the increasing functions. This is particularly annoying, considering that an alternative popular measure, taking the maximal elements on components, has the same structure as summing the values:

```
supList :: List (Float, Int) → (Float, Int)
supList = fold id f
  where f (x, n) (x′, n′) = (max x x′, max n n′)
```

The similarity of their names reflects the similarity of their implementations: both functions are folds, the only difference being the use of *max* instead of +. Nevertheless, we cannot with impunity use the generators for *supList* when testing *sumList*. Moreover, in writing more and more complicated generators, we mix up the test for the "interesting" monotonicity condition, with the "implementational" defending against overflow or round-off errors. And we still have a coverage problem, because only with knowledge of the implementation of

the measure can we estimate how well the sampling of the space of increasing functions is achieved.

It might be thought that we can always get around implementational aspects by choosing better representations for numerical values. For example, we can avoid round-off errors by replacing *Float* with rational numbers. Unfortunately, we cannot do that if the vulnerability measure requires computations which cannot be carried out on rational numbers, such as the geometric mean. Resorting to exact real numbers does not solve our problem either, because the order relation on these is not decidable, and we just trade one type of interference from the implementational aspects (defending against round-off errors) for another (guarding against undecidable comparisons).

To sum up:

- We need detailed analysis of the implementation of the function under test, and, in particular, of the datatypes they act on.
- We often need to write different custom generators even for very similar cases (such as *sumList* and *supList*).
- We mix the conceptual part of the tests with the implementational part.
- Good coverage is hard to achieve.

4 Proving correctness of vulnerability measures

It is tempting to point an accusing finger at the higher-order nature of the formalization of the vulnerability measure condition. If we hadn't used Haskell, with its functional nature and expressive type system, we might not have run into so much trouble testing the resulting implementations. Testing higher-order functions is not a topic in common textbooks on software testing [1, 20].

On the other hand, thinking about the problems we saw in the discussion of testing functoriality, it might just be that the culprit is not the exaggerated expressivity of Haskell, but on the contrary: the fact that it is not expressive enough!

In a dependently-typed programming language such as Agda, we can formulate the functor laws as types via the Curry-Howard isomorphism[3]:

```
_≐_ : {A B : Set} → (f g : A → B) → Set
f ≐ g = ∀ a → f a ≡ g a

record Functor (F : Set → Set) : Set1 where
  field
    fmap  :  {A B : Set} → (A → B) → F A → F B
    idLaw :  {A : Set} →
             fmap (id {A}) ≐ id {F A}
    compLaw : {A B C : Set} → (f : B → C) → (g : A → B) →
             fmap (f ∘ g) ≐ (fmap f ∘ fmap g)
```

Now we can also prove that the mapping function we defined is indeed functorial. The implementation of non-empty lists is virtually identical to the Haskell version:

[3] We use everywhere the propositional equality type (_≡_) provided by Agda as if it were the only equivalence relation of interest. Parameterising by different equivalence relations (using setoids instead of sets) does not introduce difficulties, but makes the examples more tedious and wastes space. Similar remarks apply to universe-polymorphism.

```
data List (A : Set) : Set where
  [_]  : A → List A
  _::_ : A → List A → List A

fold : {A B : Set} → (A → B) → (A → B → B) → List A → B
fold w c [a]        = w a
fold w c (a :: as)  = c a (fold w c as)

map : {A B : Set} → (A → B) → (List A → List B)
map f = fold ([_] ∘ f) (λ a bs → f a :: bs)
```

Proving that the map function defined preserves identities and composition is actually almost entirely performed by Agsy, the only nudging it needed was to "use the congruence of something" in the inductive step.

```
mapId : {A : Set} →
        map (id {A}) ≐ id
mapId [a]        = refl
mapId (a :: as)  = cong (λ as → a :: as) (mapId as)

mapComp : {A B C : Set} → (f : B → C) → (g : A → B) →
          map (f ∘ g) ≐ (map f ∘ map g)
mapComp f g [a]        = refl
mapComp f g (a :: as)  = cong (λ as → f (g a) :: as) (mapComp f g as)
```

Therefore, we can construct an element of type *Functor List* and clinch the proof that our *map* is a suitable choice:

```
FunctorList : Functor List
FunctorList = record {fmap    = map;
                      idLaw   = mapId;
                      compLaw = mapComp}
```

No problems with the polymorphism or higher-order nature of *map*, and, of course, no coverage problems. Motivated by this easy success, we proceed to formalize the vulnerability measure condition, starting first with the definition of increasing functions. We use the Agda standard library *IsPreorder* record for preorders, which is parameterized on the underlying equivalence (for which we use _≡_ throughout):

```
IsIncreasing : {A : Set} (_≤_ : A → A → Set) →
               (A → A) → Set
IsIncreasing (_≤_) f = ∀ a → a ≤ f a

VulnMeas : {F : Set → Set} → Functor F →
           {V : Set} → {_≤_ : V → V → Set} → IsPreorder _≡_ _≤_ →
           {W : Set} → {_⊑_ : W → W → Set} → IsPreorder _≡_ _⊑_ →
           (m : F V → W) → Set
VulnMeas {F} fF {V} {_≤_} p≤ {W} {_⊑_} p⊑ m =
  (i : V → V) → IsIncreasing _≤_ i →
                IsIncreasing _⊑m_ (fmap i)
  where fmap = Functor.fmap fF
        _⊑m_ : F V → F V → Set
        x ⊑m y = m x ⊑ m y
```

This is a virtually literal translation of Definition 1, and not more trouble to write than the *testMonotonicity* function above.

The Agda versions of our vulnerability measure candidates are also cut & paste productions from the Haskell code, except for renamings due to the lack of type classes in Agda:

```
sumList :   List (Float × Int) → Float × Int
sumList  =  fold id f
  where f : Float × Int → Float × Int → Float × Int
        f (x, n) (x′, n′)  =  (x +f x′, n +i n′)
supList  :  List (Float × Int) → Float × Int
supList  =  fold id f
  where f : Float × Int → Float × Int → Float × Int
        f (x, n) (x′, n′)  =  (maxf x x′, maxi n n′)
```

In both cases, the arguments (*id* and *f*) that *fold* receives are monotonic functions, and it is easy to see that this is a sufficient condition for a vulnerability measure. Formulating this property in Agda raises no unexpected difficulties:

```
IsMonotonous : {A : Set} → {_≤A_ : A → A → Set} → (pA : IsPreorder _≡_ _≤A_) →
               {B : Set} → {_≤B_ : B → B → Set} → (pB : IsPreorder _≡_ _≤B_) →
               (A → B) →
               Set
IsMonotonous {A} {_≤A_} pA {B} {_≤B_} pB f  =
  (a1 a2 : A) → (a1 ≤A a2) → f a1 ≤B f a2

IsMonotonous₂ : {A : Set} → {_≤A_ : A → A → Set} → (pA : IsPreorder _≡_ _≤A_) →
                {B : Set} → {_≤B_ : B → B → Set} → (pB : IsPreorder _≡_ _≤B_) →
                {C : Set} → {_≤C_ : C → C → Set} → (pC : IsPreorder _≡_ _≤C_) →
                (A → B → C) →
                Set
IsMonotonous₂ {A} {_≤A_} pA {B} {_≤B_} pB {C} {_≤C_} pC f  =
  (a1 a2 : A) → (a1 ≤A a2)  →
  (b1 b2 : B) → (b1 ≤B b2)  → f a1 b1 ≤C f a2 b2

foldMeas : {A : Set} → {_≤A_ : A → A → Set} →    (pA : IsPreorder _≡_ _≤A_) →
           {B : Set} → {_≤B_ : B → B → Set} →    (pB : IsPreorder _≡_ _≤B_) →
           (w : A → B)        → IsMonotonous  pA pB      w →
           (c : A → B → B) → IsMonotonous₂ pA pB pB c  →
           VulnMeas FunctorList pA pB (fold w c)
```

Folding monotonic functions over non-empty lists produces vulnerability measures: how hard is it to convince the type checker of this fact? Perhaps surprisingly, not hard at all. Agsy finds out all by itself that increasing the elements of a singleton list and applying a monotonic function to the result is going to result in an increased measure:

```
foldMeas pA pB w monw c mon₂c i isInc [a]  =  monw a (i a) (isInc a)
```

More impressively, in the inductive case, after the gentle nudge to apply the monotonicity of the second argument to fold, Agsy can fill in all the arguments to *mon₂c* except for the last one, the induction hypothesis:

```
foldMeas pA pB w monw c mon₂c i isInc (a :: as)  =
  mon₂c a (i a) (isInc a)
        (fold w c as)
        (fold w c (fold (λ x → [i x]) (λ x → _::_ (i x)) as))
        ?
```

which we fill in and, after tempering a bit Agsy's eagerness to reduce every term to normal form, we reach the final version:

```
foldMeas pA pB w monw c mon₂c i isInc (a :: as) =
  mon₂c a              (i a)                  (isInc a)
        (fold w c as) (fold w c (map i as)) (foldMeas pA pB w monw c mon₂c i isInc as)
```

All that remains to do in order to ensure that our candidates, *sumList* and *supList* are indeed vulnerability measures is to prove the monotonicity of $id, +_f, +_i, maxf, maxi$. Well, we cannot! *Float* and *Int* are machine built-in types, which Agda allows us access with a bit of builtin-trickery:

```
postulate Float : Set {-# BUILTIN FLOAT Float #-}
primitive
  primFloatPlus : Float → Float → Float
  primFloatLess : Float → Float → Bool
_+_ : Float → Float → Float
_+_ = primFloatPlus
_≤_ : Float → Float → Bool
_≤_ = primFloatLessThan
```

And the same thing again for *Int*. But, beyond the signature of these functions, the type checker knows nothing about them, and any additional property must be postulated, for example:

```
postulate ≤fRefl  : (x : Float) → x ≤f x
postulate ≤fTrans : (x y z : Float) →
                    x ≤f y → y ≤f z → x ≤f z
postulate +fmon   : (x y x' y' : Float) →
                    x ≤f x' → y ≤f y' →
                    (x +f y) ≤f (x' +f y')
```

where $\leqslant f$ is a suitably lifted representation of the primitive boolean relation. The type checker accepts then (but does not guarantee) that these properties hold, and we obtain thus a conditional proof of correctness, with the implementational aspects nicely tucked away and signalled by the **postulate** keyword.

Alternatively, we can use Peano naturals instead of *Int* and rationals instead of *Float*, for which we *can* prove the required properties, and obtain an unconditional result (and a less efficient program). Eventually, one expects such properties to be part of standard libraries, and have an even easier time switching from one datatype to another. In any case, the most difficult part of the job, proving that a fold gives a vulnerability measure, is independent of the specific datatype considered.

To sum up, formulating the vulnerability measure condition via the Curry-Howard isomorphism is not more difficult than coming up with the corresponding tests, while proving it for the cases we considered is easier and more general than implementing those tests. The conceptual and implementational aspects are cleanly separated, and the problematic spots highlighted by the **postulate** keyword.

5 Conclusions

There have been several papers lately that show the advantages of dependently-typed programming languages for embedded domain-specific languages [5, 19, 24], and we have just provided another example.

A feature that distinguishes our work from the others is that it brings us in contact with scientific programming: the kind of programming that covers the models used to generate the possible trajectories to be measured. The scientific programming community often tackles problems with the sort of features our example illustrates, where exhaustive testing is not feasible and formal proofs of correctness might be easier. Scientific programmers tend also to be familiar with mathematical proof in an informal context: many numerical methods are justified by some sort of informal proof of correctness, which is then a candidate for translating to a formal context. The question therefore is, why is formal proof not used more frequently in scientific programming?

One reason is probably that usable implementations of dependently-typed programming languages have not been around very long. Moreover, the experience we have accumulated with them has been more on the discrete, algebraic side and rather less on the continuous, real analysis side which is important for scientific programming. The Agda standard library [26], young as it is (currently at version *0.6*), implements many kinds of algebraic structures, but has no mention of the *Float* datatype or real numbers. There are, to our knowledge, no dependently-typed libraries available for doing the sort of things that a scientific programmer takes for granted: solving linear systems, factorizing matrices, interpolating real functions, optimization, and so on.

Developing such libraries in a dependently-typed programming language is quite challenging. Consider, for example, that in order to implement an optimization method, one has to specify exactly what is meant by "optimization": does the method return the exact solution or just an approximation of it?

We can attempt to obtain the exact solution if we work with constructive real numbers in the realm of constructive real analysis, as suggested, for example, by Bishop [3]. There are several representations of exact real numbers: the ones most used in constructive numerical analysis are based on the work of Russell O'Connor in Nijmegen [22, 23]. Validated numerical methods via constructive analysis is still a research subject. There are promising results [12], but they are quite far from providing a usable basis for scientific programming. In particular, there are no library functions available yet for solving a linear system of equations.

An alternative approach is to content ourselves with an approximate solution. After all, the vast majority of numerical libraries available today work with floating point numbers and thus abandon the search for an exact solution from the beginning. Here the challenge is to specify what *is* being computed: what guarantees are made about the quality of the approximation delivered? Existing libraries tend to be surprisingly vague here, encouraging a trial-and-error approach and relying on the expertise of the user. The arguments for why a certain method should lead to a good approximation of the solution are also often expressed in terms of exact real numbers and therefore can only be formalized with the help of postulates, as we have done above.

To do better, one has to formalize the properties of floating-point numbers as expressed in the IEEE 754 or 854 floating-point arithmetic standard. Several such formalizations have been achieved in PVS [18], HOL [10], and Coq [4], and have been used to verify the implementation of algorithms for fundamental and relatively simple functions, such as the square root or the exponential. To our knowledge, no substantial numerical methods have

yet been verified. Moreover, this kind of work is hard to do in an academic context, and we might have to wait until industry is motivated enough to fund it.

Until such a time, the best that we can do is to separate the problems that require the continuous / analytic from those that deal more with the discrete / algebraic, and prove the correctness of the latter conditional on (postulated) correctness of the former, which we can at most test. In this sense, in the above examples, we were indeed lucky, having to deal only with algebraic structures such as preorders and lists, and being satisfied with correctness conditioned on the field structure of floating-point numbers and integers (a structure they, in fact, do not have!).

Acknowledgement

The authors gratefully acknowledges the fruitful discussions with Andreas Abel, Jean-Philippe Bernardy, Paul Flondor, and the members of the Cartesian Seminar at PIK.

References

1 P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

2 Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. Testing polymorphic properties. In Andrew Gordon, editor, *European Symposium on Programming*, volume 6012 of *LNCS*, pages 125–144. Springer, 2010.

3 E. Bishop and D. Bridges. *Constructive Analysis*. Springer, New York, 1985.

4 S. Boldo and G. Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *Proceedings of the 20th IEEE Symposium on Computer Arithmetic*, pages 243–252, Tübingen, Germany, July 2011. IEEE CS Press.

5 Edwin C. Brady. Idris — systems programming meets full dependent types. In *Proc. 5th ACM workshop on Programming languages meets program verification*, PLPV '11, pages 43–54. ACM, 2011.

6 T. Cannon, J. Twigg, and J. Rowell. Social vulnerability, sustainable livelihoods and disasters. Technical report, AON Benfield UCL Hazard Research Center, 2002. Available at `http://www.abuhrc.org/Documents/Social_vulnerability_sust_live.pdf`.

7 J. Chakraborty, G.A. Tobin, and B.E. Montz. Population evacuation: assessing spatial variability in geophysical risk and social vulnerability to natural hazards. *Natural Hazards Review*, pages 22–33, February 2005.

8 K. Claessen and J. Hughes. Specification based testing with QuickCheck. In *The Fun of Programming*, Cornerstones of Computing, pages 17–40. Palgrave, 2003.

9 T. Feldbrügge and J. von Braun. Is the world becoming a more risky place? Trends in disasters and vulnerability to them. Technical Report 46, Center for Development Research, Bonn, 2002.

10 John Harrison. Floating point verification in HOL Light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997. Available as `http://www.cl.cam.ac.uk/~jrh13/papers/tang.html`.

11 C. Ionescu. *Vulnerability Modeling and Monadic Dynamical Systems*. PhD thesis, Department of Mathematics and Informatics, Free University Berlin, February 2009.

12 R. Krebbers and B. Spitters. Computer certified efficient exact reals in Coq. In James H. Davenport et al., editors, *Calculemus/MKM*, volume 6824 of *LNCS*, pages 90–106. Springer, 2011.

13 D. Lemmen and F. Warren, editors. *Climate Change Impacts and Adaptation: A Canadian Perspective*. Natural Resources Canada, 2004.

14 Daniel Lincke, Patrik Jansson, Marcin Zalewski, and Cezar Ionescu. Generic libraries in C++ with concepts from high-level domain descriptions in Haskell: A DSL for computational vulnerability assessment. In Walid Taha, editor, *IFIP Working Conf. on Domain Specific Languages*, volume 5658 of *LNCS*, pages 236–261, 2009.

15 F. Lindblad and M. Benke. A tool for automated theorem proving in Agda. In Jean-Christophe Filliâtre et al., editors, *Types for Proofs and Programs, International Workshop, TYPES 2004*, LNCS, pages 154–169. Springer, 2006.

16 J.J. McCarthy, O. Canziani, N.A. Leary, D.J. Dokken, and K.S. White, editors. *Climate Change 2001: Impacts, Adaptation and Vulnerability. Contribution of Working Group II to the Third Assessment Report of the Intergovernmental Panel on Climate Change*. Cambridge University Press, 2001.

17 M.J. Metzger and D. Schröter. Towards a spatially explicit and quantitative vulnerability assessment of environmental change in Europe. *Regional Environmental Change*, 6(4):201–216, 2006.

18 Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical report, 1995. Technical Memorandum 110167, NASA, Langley Research. Availble as `http://nasa1995.tpub.com/NASA-95-tm110167/`.

19 Jamie Morgenstern and Daniel R. Licata. Security-typed programming within dependently typed programming. In *Proc. 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 169–180. ACM, 2010.

20 G.J. Myers. *The Art of Software Testing. Second edition, revised and updated by T. Badgett and T.M. Thomas with C. Sandler*. John Wiley & Sons, Inc., 2004.

21 U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, September 2007.

22 R. O'Connor. A monadic, functional implementation of real numbers. *Mathematical Structures in Computer Science*, 1:129–159, 2007.

23 Russell O'Connor. Certified exact transcendental real number computation in Coq. In Otmane Ait Mohamed et al., editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *LNCS*, pages 246–261. Springer, 2008.

24 N. Oury and W. Swierstra. The power of Pi. In *International Conference on Functional Programming*, pages 39–50. ACM, 2008.

25 B. Pascal. *Minor Works, translated by O. W. Wright*, volume XLVIII, Part 2 of *The Harvard classics*. P.F. Collier & Son, 1909-14.

26 The Agda Team. The Agda Wiki, 2011. Available from `http://wiki.portal.chalmers.se/agda/pmwiki.php`. Includes documentation, links to the Agda implementation and to the standard library.

27 K. Thywissen. Components of risk, a comparative glossary. *SOURCE - Studies Of the University: Research, Counsel, Education*, 2, 2006.

28 UN/ISDR (United Nations International Strategy for Disaster Reduction). *Living with Risk. A Global Review of Disaster Reduction Initiatives*. United Nations, Geneva, 2004.