Dependently-typed programming in scientific computing

Cezar Ionescu[1] and Patrik Jansson[2]

[1] Potsdam Institute for Climate Impact Research
[2] Chalmers University of Technology

Abstract. Computer simulations are essential in virtually every scientific discipline, even more so in those such as economics or climate change where the ability to make laboratory experiments is limited. Therefore, it is important to ensure that the models are implemented correctly, that they can be re-implemented and that the results can be reproduced. Typically, though, the models are described by a mixture of prose and mathematics which is insufficient for these purposes. We argue that using dependent types allows us to gradually reduce the gap between the mathematical description and the implementation, and we give examples from economic modelling. We discuss the consequences that our incremental approach has on programming style and the requirements it imposes on the dependently-typed programming languages used.

1 Introduction

In 2006, Herbert Gintis [10] announced the discovery of a mechanism that would explain price formation and disequilibrium adjustment without requiring the presence of a central authority or omniscience on part of the agents, as is currently assumed in mainstream economics. Gintis' results were, as he put it "empirical rather than theoretical: we have created a class of economies and investigated their properties for a range of parameters." They were obtained by computer simulations. Due to the importance of this result, two groups of researchers, one at PIK, the other at Chalmers [9], independently attempted to do something which should perhaps be routine, but is hardly ever done: to re-implement the model described in the paper and reproduce the results. After initial attempts failed and Gintis graciously provided the source code, both groups discovered several ways that his implementation diverged from the description in the paper, only one of which could be called a "bug". Much more problematic was the ambiguity left open by the model description given in the paper, which consisted of a mixture of prose and mathematical equations.

The example of the Gintis model was chosen because it is well documented in recent literature, not because it is unique. It is quite typical for scientists to believe that the mathematical equations used to develop a model are sufficient specification for the implementation of that model, but that is rarely the case. Discretizations, approximations, choices of integration methods, and many other similar steps come between the mathematical description and the program.

This is a gap that must be bridged if we are to be able to check correctness of implementations, re-implement models, or replicate results.

Sooner or later, everyone who considers this problem is bound to encounter constructive mathematics and Martin-Löf's type theory, which seems to be made to order for this purpose. Here is for example a quote from the programmatic article "Constructive Mathematics and Computer Programming" [12]:

> Now, it is the contention of the intuitionists (or constructivists, I shall use these terms synonymously) that the basic mathematical notions, above all the notion of function, ought to be interpreted in such a way that the cleavage between mathematics, classical mathematics, that is, and programming that we are witnessing at present disappears.

Specifications ("tasks that the programs are supposed to perform") are also mentioned explicitly:

> [Type theory] provides a precise notation not only, like other programming languages, for the programs themselves but also for the tasks that the programs are supposed to perform. Thus the correctness of a program written in the theory of types is proved formally at the same time as it is being synthesized.

The ideal of correctness put forward here is very enticing. There are many examples of such correct-by-construction development, for example [16,14,20,19,7]. It seems natural to attempt to apply the same methodology in the context of scientific computing, for instance when building economic models such as the one we mentioned in the beginning.

A necessary (but far from sufficient) condition for that is that type theory has the expressive power to formulate the usual mathematical concepts which modelers use as specifications. In the next section we show that this is indeed the case. Together with economists at PIK, we have formalized basic building blocks of economic theory, used in almost all economic models today, concepts such as Pareto efficiency, Walrasian equilibrium, Nash equilibrium, and a host of others, together with the relations between them (for example, Walrasian equilibria are Pareto efficient). The resulting formalizations are pleasantly close to the mathematical formulations the modelers are used to, so we can hope they could use them in specifications.

The bad news is most of these concepts are classical in nature: economics is currently a non-constructive theory (and even the so-called "computable general equilibrium models" turn out to be non-computable). Therefore, the specifications turn out to be non-implementable and the gap between the mathematics and the programming is still there. But, as we argue in the third section, we are now in a better position to close it.

We close the paper with a discussion of some consequences of this approach.

2 Formalizing economic notions in type theory

The quintessential economic situation is that of exchange of goods, which we introduce via the simplest possible example: two agents and two goods. We have to assume at least two agents and two goods: if we had only one agent there would be no one to exchange with, and if there were only one kind of good then there would be nothing against which to exchange that good. We would then have a situation of gift-giving, rather than exchange.

For concreteness, let us call the first good "wine" and the second good "beer", assume they come in bottles and cans respectively, and that there are 5 bottles of wine and 10 cans of beer, distributed among our two agents: agent one has 3 bottles of wine and 3 cans of beer and agent two has 2 bottle of wine and 7 cans of beer. The bundle of goods each agent has is called its *endowment*, the distribution of the endowments is an *allocation*.

Let us assume that the agents have different preferences for beer and wine. For example, agent one likes beer more than wine, but needs to have at least one bottle of wine in case he has more sophisticated guests. Agent two, on the other hand, values wine over beer, but must have at least three cans of beer for watching football with friends. The agents are allowed to change their endowments by trading, but in the end there must be exactly as many bottles and cans as we started out with: there is no consumption and no production of goods, only pure exchange.

In our example, agents have preferences over their endowments (their own stocks only), but in general they could have them over allocations (including their competitors' endowments), allowing economists to model not just greed, but also envy. In most common examples, preferences are total preorders (reflexive and transitive, but not necessarily anti-symmetric). An agent's preference over endowments can be extended to preference over allocations in the natural way (by just ignoring others' endowments).

An exchange leads to a re-allocation of goods, but the resulting allocation must be *feasible*: this includes the "no creation, no consumption"-condition, but also the constraints the agents have (at least one bottle of wine for the first agent and three cans of beer for the second one).

Under the assumptions we have made, we can expect that the two agents will indeed trade with each other, since each one stands to gain by an exchange. This would not be the case if we switched the two preferences (or, equivalently, the two endowments) because then the agent who prefers wine would not be able to trade any of his beer for it because of the 3-cans constraint. Coming back to the original setting, we can also see that intuitively a re-allocation of goods in which agent one has 1 bottle of wine and 7 cans of beer (and the rest goes to agent two) is optimal. The two agents are as well off as they can possibly be, given their initial endowments and their preferences. An allocation in which agent one has 2 bottles of wine and 7 cans of beer (and agent two therefore 3 and 3) is also feasible and preferred by both agents to the initial one, but is intuitively less satisfactory. Still, it is a possible end-result of an exchange between the agents.

The reader should now be in a position to understand the following definitions taken from the standard textbook on microeconomics:

> **Definitions of Pareto efficiency.** A feasible allocation $\boldsymbol{x}$ is a **weakly Pareto efficient** allocation if there is no feasible allocation $\boldsymbol{x'}$ such that all agents strictly prefer $\boldsymbol{x'}$ to $\boldsymbol{x}$. A feasible allocation $\boldsymbol{x}$ is a **strongly Pareto efficient** allocation if there is no feasible allocation $\boldsymbol{x'}$ such that all agents weakly prefer $\boldsymbol{x'}$ to $\boldsymbol{x}$, and some agent strictly prefers $\boldsymbol{x'}$ to $\boldsymbol{x}$.
>
> *Varian [23], p. 323*

In our example, the first re-allocation, intuitively considered optimal, can be seen to be strongly Pareto efficient, while the second one, less satisfactory, but not leading necessarily to an exchange, is weakly Pareto efficient.

Pareto efficiency is fundamental in economics and easily formalized in constructive type theory, which makes it a good place to start. Since our economist colleagues were familiar with Haskell, we chose to work with implementations of type theory which offer a similar syntax, so we have used equally Agda and Idris (here we present the Agda version).

We were fortunate that we could assume familiarity with a functional programming language, which is not currently part of the standard training of economists. We were even more fortunate that we could assume familiarity with the ideas and practice of formalization, at the level of, for example, Chapter 12 of Suppes' *Introduction to Logic* [18]. The interdisciplinary nature of research at PIK, involving a mixture of natural and social sciences, has led to many inquiries into the meaning of words such as "sustainability", "resilience", or "vulnerability" in the context of climate change. There have been a number of projects, workshops, and seminars devoted to the topic of formalization and mathematical modeling of such concepts using classical logic and set theory.

Accordingly, our formalization of Pareto efficiency has a distinctively set-theoretical flavor. We assume a set Agent for the agents, a set Allocation for the allocations, a predicate Feasible on this set, and a ternary relation of strict preference. In Agda, the standard way of working with such assumptions is to pass them as parameters to the module encapsulating the formalization. Alternatively, we can explicitly express them as postulates:

```
postulate
  Agent      : Set
  Allocation : Set
  Feasible   : Allocation → Set
  _strictlyPrefers_to_  : Agent → Allocation → Allocation → Set
```

The formalization of weak Pareto efficiency as a predicate on allocations reads

```
WeakPareto x  =  Feasible x ∧
  ¬ (∃ [x' : Allocation] (Feasible x' ∧
                        (∀ [a : Agent] (a strictlyPrefers x' to x))))
```

We are using here Agda's flexible, Unicode-enabled syntax, to make the formalization readable to anyone familiar with the standard logical connectors and quantifiers. It is, we hope, clearly an ad-litteram translation of the definition cited above. To achieve this effect, we have sometimes used a different notation than that of the standard Agda library, for example we use ∃ where the standard library has Σ. The most important departures from the standard are noted in the Appendix, which also lists references for readers unfamiliar with Agda or the monomorphic version of Martin-Löf's type theory it implements.

The formalization of strong Pareto efficiency requires an additional ternary relation for weak preference, but is otherwise just as simple:

```
postulate
  _weaklyPrefers_to_  : Agent → Allocation → Allocation → Set
StrongPareto x = Feasible x ∧
  ¬ (∃ [x' : Allocation] (Feasible x' ∧
                          (∀ [a : Agent]  (a  weaklyPrefers x' to x)) ∧
                          (∃ [a' : Agent] (a' strictlyPrefers x' to x))))
```

It is just as easy to formulate a simple relationship between weak and strong Pareto efficiency: namely, that strong Pareto efficiency is stronger than weak Pareto efficiency, i.e., the former implies the latter:

```
Strong=>Weak : ∀ [x : Allocation] (StrongPareto x → WeakPareto x)
```

but this is as far as we can go without discussing the meaning of the connectives and quantifiers.

Until now, the formulas we have seen could have been written in classical logic. Typed predicate logic, for example, introduced by Raymond Turner in [22], has the same syntax as constructive type theory, but is a classical, multi-sorted predicate logic. What is different is the inferential system: what counts as a proof.

Constructively, the universal quantifier above is interpreted as a function which, to each allocation x, associates a proof of the statement "x is strongly Pareto efficient implies that x is weakly Pareto efficient". In turn, this implication is interpreted as a function which, given a proof that x is strongly Pareto efficient, produces a proof that x is weakly Pareto efficient. A proof that x is weakly Pareto efficient consists of a pair of proofs: one that x is feasible, the other that it is impossible to find an x' which is also feasible and strictly preferred by all agents to x. And so on.

It takes a bit of getting used to, but after that, and with a little help from the Agda proof assistant Agsy, it is easy to implement proofs such as the following:

```
postulate
  agent0       : Agent
  strict=>weak : ∀ {a x x'} → a strictlyPrefers x' to x →
                              a weaklyPrefers x' to x
Strong=>Weak x (fx, spx) = (fx, wpx)
```

```
where
wpx  :  ¬ (∃ [x' : Allocation] (   Feasible x' ∧
                                    (∀ [a : Agent] (a strictlyPrefers x' to x))))
wpx (x', (fx', prefx'))  =
   spx (x', (fx', ((λ a → strict=>weak (prefx' a)),
                  (agent0, prefx' agent0))))
```

In fact, it is instructive to do so. Here, we can see that we need the assumption that strict preferences imply weak preferences (which holds in the common model of preferences as total preorders) and that the set of agents is not empty (and that we can actually pick an agent from it, whom we called agent0). As is often the case, assumptions are made explicit by formalization.

From Pareto efficiency we move on to one of the most important notions of economics: that of a Walrasian equilibrium.

In a first approximation, Walrasian equilibrium can be understood as a way of computing Pareto efficient allocations. This computation is difficult in general, among other reasons because it involves looking at all the agents simultaneously. It would be much easier if the agents could somehow be treated individually, instead of collectively.

In the model proposed by Walras in [27], goods have prices. Since each agent starts out with an initial endowment, the value of this endowment can be computed to yield the agent's budget. Each agent then computes the optimal endowment within this budget. Suppose these optimal endowments together make up a feasible allocation: this would surely be a good end-result for an exchange, since every agent gets the best it can afford. Whether this optimal allocation is in fact feasible depends on the prices. In our example above, if the price of a can of beer were the same as the price of a bottle of wine, say 1 cent each, then agent one would have as optimal endowment one bottle of wine and five cans of beer, while agent two could optimally afford six bottles of wine and three cans of beer. The resulting allocation is not feasible: it needs too many bottles of wine (seven, instead of the five available ones) and too few cans of beer (eight, instead of ten). On the other hand, if wine bottles cost twice as much as beer cans, say two cents to one, then the optimal endowments of the two agents make up the strongly Pareto allocation we have seen earlier. Prices for which the optimal endowments constitute a feasible allocation are called *equilibrium prices*, together with an optimal allocation they make up a *Walrasian equilibrium.* Here is the definition from Varian's classical textbook [23]:

> An allocation-price pair $(\boldsymbol{x}, \boldsymbol{p})$ is a **Walrasian equilibrium** if (1) the allocation is feasible, and (2) each agent is making an optimal choice from its budget set. In equations:
>
> 1. $\sum_{i=1}^{n} \boldsymbol{x}_i = \sum_{i=1}^{n} \boldsymbol{\omega}_i$
> 2. If $\boldsymbol{x'}_i$ is preferred by agent i to $\boldsymbol{x}_i$, then $\boldsymbol{p}\boldsymbol{x'}_i > \boldsymbol{p}\boldsymbol{\omega}_i$.
>
> Varian, *Microeconomic Analysis*, p. 325

Here, $\boldsymbol{\omega}$ is the initial allocation. It is assumed that the endowments are vectors of non-negative real numbers, each component representing the quantity of the respective good, and that the value of an endowment is computed by a scalar product with the prices. In turn, an allocation is represented by a matrix, having the individual endowments as columns. Thus, the sum in the first point in the definition is column-wise and represents the conservation of goods condition. The second point states that if an allocation is preferred by an agent to the optimal one, the value of the agent's endowment in this allocation is greater than the value of the agent's initial endowment: the allocation is out of budget.

There is a level of detail in this definition which is more than we need for the moment. For the purpose of formalizing Walrasian equilibrium, it suffices to assume that we can compute an agent's endowment from an allocation and the value of that endowment at given prices, and that we can compare values with one another. This being granted, the precise nature of the sets of prices, endowments, values is not important and we can formalize a more general version of Walrasian equilibrium:

```
postulate
  Endowment  : Set
  Price      : Set
  Value      : Set
  endmt      : Allocation → Agent → Endowment
  value      : Endowment → Price → Value
  _>_        : Value → Value → Set
  ω          : Allocation
WalrasianEq (p, x) = Feasible x ∧
  (∀ [a : Agent] (∀ [x' : Allocation]
    ((a strictlyPrefers x' to x) →
      value (endmt x' a) p > value (endmt ω a) p)))
```

While a bit more abstract, this formalization is still just an almost literal translation of the definition given by Varian.

We have referred to the allocation in a Walrasian equilibrium as "optimal", but is it really Pareto efficient? In fact, it is easy to show that it is weakly Pareto efficient, a result known as the first theorem of welfare economics:

> **First theorem of welfare economics.** If $(\boldsymbol{x}, \boldsymbol{p})$ is a Walrasian equilibrium, then x is [weakly] Pareto efficient.
>
> Varian, *Microeconomic Analysis*, p. 326

We have explicitly added the qualifier *weakly*: Varian adopts the following convention "when we say 'Pareto efficient' we generally mean 'weakly Pareto efficient' " (p. 324).

The proof of the theorem is by contradiction and relies on the distributivity of multiplication over addition, on factor cancellation and on the assumption that prices are strictly positive (and therefore non-zero). We can abstract away from these properties by postulating that, for any prices p and any allocation x, if every agent's endowment in x is more valuable than in ω, then x is not feasible:

```
postulate
  outOfBudget : ∀ [p : Price] (∀ [x : Allocation]
    (∀ [a : Agent] (value (endmt x a) p > value (endmt ω a) p) →
    ¬ (Feasible x)))
```

The formalization of the theorem is then short and simple, at least if one is accustomed to the computational reading of the logical connectives and quantifiers.

```
FirstTheorem : ∀ [p : Price] (∀ [x : Allocation] (WalrasianEq (p, x) →
                                                   WeakPareto x))
FirstTheorem p x (fx, weq) = (fx, wpe) where
  wpe : ¬ (∃ [x' : Allocation] (Feasible x' ∧
                               (∀ [a : Agent] (a strictlyPrefers x' to x))))
  wpe (x', (fx', prefx')) = outOfBudget p x'
                                      (λ a → weq a x' (prefx' a)) fx'
```

More interestingly is that, while formalizing this proof, we hit upon the following question: if $(\boldsymbol{x}, \boldsymbol{p})$ is a Walrasian equilibrium, is then every endowment in $\boldsymbol{x}$ in the respective agent's budget? The answer, which even some of our economist colleagues found surprising, is no. Of course, if one has the idea of looking for them, counter-examples are easy to find. Consider our two-agent example, with the same initial allocation, but removing the constraints on the preferences: agent one no longer needs to have at least one bottle of wine, and agent two no longer cares about beer.

The former equilibrium prices, two cents for a bottle of wine and one for a can of beer, are still equilibrium prices for the new situations. The allocation which gives agent one an endowment of no wine and nine cans of beer, and agent two all five bottles of wine and one can of the beer is optimal. Any allocation strictly preferred by agent one would have more cans of beer than it can afford, and the same for agent two in terms of wine (no half-bottles accepted!). Therefore, the Walrasian equilibrium condition is satisfied.

Now consider the allocation that gives agent one all ten cans of beer, and agent two the five bottles of wine. This allocation is certainly feasible: it contains five bottles of wine and ten cans of beer, just like ω. If another allocation is preferred by agent one, it has to give it at least ten cans of beer: more than it can afford. If it is preferred by agent two, it has to give it six bottles of wine, but agent two can only afford five. Therefore, this allocation is also a Walrasian equilibrium for these prices. But, as we see, it is out of budget for agent one.

We have said that Walrasian equilibrium can be approached as a way of computing Pareto efficient allocation, an idea that we found useful but which might make some of our economist colleagues cringe. Before going further, we should point out that the importance of the Walrasian model lies in that it serves to explain prices as arising from desirability of goods, from the preferences of the agents and their initial endowments, as opposed to the Marxist theory of value, where prices appear as a measure of the labor involved in the production of goods.

This model admits many extensions: one can add to it production and consumption of goods, a labor market (treating labor as a good to be exchanged by the workers), exchanges in several steps (adding a temporal dimension to the problem), and so on. Most mainstream economic models, including such as are used for policy advice (for example ReMIND [4] and GEM-E3 [3]) are *general equilibrium models* based on extensions of the Walrasian ideas formalized in this section.

A general criticism of all these models is that they neglect the dynamical aspect of reaching the equilibrium situation. There is no known plausible mechanism which explains exactly how equilibrium prices can arise in practice. Walras' own proposal for such a price-formation mechanism involved an *auctioneer*. This is a central entity who can see all supply-demand imbalances and adjust prices accordingly, raising the prices of goods for which there is too great demand, and lowering those for which there is too little, in an iterative process. Even if one accepts that in some situations one could have an authority that might act as auctioneer, there is no general proof that the iterative process will eventually converge.

This, in fact, was the problem that Gintis attempted to solve, and the reason the papers we referred to in the previous section found an immediate echo in the economics community. This shows that even non-mainstream models like his can actually benefit from having formal specifications of the classical economic concepts.

We have formalized much more than just what we have shown here: the detailed definitions in Varian's book, but also the notions of Nash equilibria, correlated equilibria, and several others. They are all more complex, but not more complicated than what we have been able to show here. All in all, we can say that constructive type theory as embodied by Agda or Idris has passed our test for expressiveness: we were able to formulate in it fundamental notions of economics and relationships between them in such a way that they can be read, and with some exercise even used, by our colleagues.

That was the good news. The bad news is that most of these concepts are not constructive. Specifications of programs that take as input agents characterized by preference relations and initial endowments and return a Walrasian (or Nash, or correlated, ...) equilibrium can in general not be fulfilled. Even the so-called computable general equilibrium models are not, in fact, computable [25].

We started with the problem that the mathematical descriptions employed in scientific computing are too far from the implementation to serve as specifications. Constructive type theory promised to be a bridge across this gulf. However, it now appears that we have not made any progress. The simplicity of the translation from informal mathematical definitions to formal ones, which we interpreted as proof of the expressiveness of constructive type theory, turns out to have been deceptive. We appear to have the same unsatisfactory specifications, only in slightly fancier notation. The gap has not been bridged, after all.

However, the translational effort has brought us something essential, as we shall see in the next section.

3 Increasingly correct scientific computing

There are general reasons for wanting to formalize the kind of mathematical specifications used in scientific programming. For one thing, formalization can help us understand the informal definitions better. We have seen this in the case of Walrasian equilibrium, where optimal allocations are allowed by the standard definition to be "out of budget" for some agents. For another, having formal specifications makes them checkable by computer. We can be fairly confident that syntactic errors are going to be signalled by the type checker, as well as some of the more glaring semantical errors, such as inverting quantifier order.

Still, why choose constructive type theory as the vehicle for the formalization, over, for example, classical higher-order logic and set theory (which also have the advantage of being more familiar to non-computer scientists)?

The reason is that the only way one can decrease the distance between mathematics and programming is to make the mathematical side more constructive: computation cannot become more classical. There are many efforts underway aiming to use constructive mathematics in the context of scientific computing. For example, Velupillai's program for computable economics [24,26], or various projects for developing constructive numerical methods [2,11].

Using constructive type theory both for formalizing specifications and for the implementations enables us to take advantage of these developments as they occur, by gradually replacing the non-constructive concepts with their constructive counterparts. This is an increase of correctness "from above": we are improving the specifications, becoming more precise about what we are "really" computing and what relationship there is between this and what we think we should be computing.

We can already implement the constructive parts, and isolate the ones that depend on classical theory in *postulates*. For example, the kind of inter-temporal optimization that many economic models are based on can be solved by applying Bellman's dynamic programming algorithm [5,6], thus reducing the inter-temporal optimization to the successive application of local optimizations. This works if the Bellman principle can be applied, and the proof is constructive. Thus, we can have a verified implementation of the dynamic programming algorithm, *if* we can implement the local maximization.

Few modelers are going to implement their own optimization routine. Rather, they are going to use an external one, with an interface which in its simplest form can be expressed as

```
maxUtil : {n : Nat} → (Vect Float n → Float) → Float
```

so that maxUtil u returns the maximum of the utility function u defined over $\mathbb{R}^n$. (For brevity, we ignore here that a function such as maxUtil should also return the input vector for which the utility reaches its maximum). The modeler will often use maxUtil as if it implemented the specification

```
postulate maxSpec : {n : Nat} →
  ∀ [u : (Vect Float n → Float)]
    (∀ [x : Vect Float n]
      (so (u x ⩽ maxUtil u)))
```

In this usage, postulates express a condition relative to which the correctness of the implementation is to be judged. In particular, this is always the case when using external routines which do not have a type theoretical interface. The typechecker can at least verify that we are using the postulated properties in a correct way. Another advantage is that we have clearly signalled the spots where further refinement is necessary, where constructive mathematics can help.

There is another usage of postulates which points to further refinements in a different, simpler way. The scientists involved in the modeling process are usually not experts in giving formal proofs, let alone constructive formal proofs. Sometimes, it is useful to just defer the proof to the experts, or to a later stage of development. For example, while maximizing utility functions is in general not computable, it is computable when the domain of the utility is a finite set. Therefore, the following specification

```
postulate maxFinSpec : {n : Nat} →
                         ∀ [u : (Fin (S n) → Float)]
                           (∀ [i : Fin (S n)]
                             (so (u i ⩽ maxUtil u)))
```

is implementable, but the proof might be tricky for the beginner.

In fact, beginners, even under the somewhat ideal conditions of familiarity with Haskell, tend to paint themselves in a corner. For example, to implement maximization by enumeration, one might try to translate the following Haskell code:

```
maxUtil :: Nat -> (Nat -> Float) -> Float
maxUtil 0       u  =  u 0
maxUtil (n + 1) u  =  maxUtil' (n + 1) u (u 0) 0

maxUtil' :: Nat -> (Nat -> Float) -> Float -> Nat -> Float
maxUtil' n u best c' =
  let c   =  c' + 1 in  --  c is the candidate new best
  let uc  =  u c    in  --  uc is potential new optimum
  let bU  =  max uc best in
      if  c == n        -- is cand the last candidate?
          then bU
          else maxUtil' n u bU c
```

in the following somewhat exaggeratedly literal manner. We hasten to say that we are not presenting this example as a model of good style, instead, it's main merit is that it has actually arisen in practice and is a somewhat typical and instructive case:

```
maxUtil : {n : Nat} → (Fin (S n) → Float) → Float
maxUtil {O} u    = u fO
maxUtil {S n} u  = maxUtil' u (u fO) fO
maxUtil'  :  {n : Nat} →
             (Fin (S n) → Float) → Float → Fin n → Float
maxUtil' {n} u bestU c'  =
  let c   = fS c' in       -- c is the candidate
  let uc  = u c in         -- uc is potential new optimum
  let bU  = max uc bestU in
    if (c =F= toFin n)     -- is cand the last candidate?
      then bU
      else (maxUtil' u bU c)   -- !
```

But this code does not type check! The reason is that the type of the last argument to maxUtil', namely c, is Fin (S n) instead of Fin n, as required by our use of it (namely to increment it in order to obtain a new candidate). We know that, in fact, c could be cast to a valid value of type Fin n, since it we have just tested that it is not maximal in Fin (S n), but we are going to have a very hard time convincing the type checker of it, let alone prove that the resulting max satisfies maxFinSpec. On top of it all, the termination checker also complains it does not see why maxUtil' is not just going to loop forever. Unfortunately, this situation is quite common when attempting to just write Haskell in Agda.

At this point, many a scientific programmer can feel like throwing their hands up and returning to Haskell or Fortran. Which is why it is important that the language provides some form of unsafe cast. In Agda, this takes the form of trustMe, on the basis of which one can write a version of the Haskell function unsafeCoerce (see the Appendix) which can be used to eliminate the type error. We can get the above code to work by replacing the last line with else (maxUtil' u bU (coerce c)) and adding a no-temination-check option for the compiler.

But, in so doing we have lost all additional safety provided by the dependently-typed system: we are just writing Haskell in Agda. Again, the same question arises: why not just write Haskell then? And, again, the same answer: because here we can improve. It is an instructive exercise to repair the maxUtil function, eliminating the unsafe elements, while still keeping it tail-recursive. In doing so, one discovers that the main culprit is the boolean test, where work is done to determine if the candidate is maximal, only to immediately discard that work so that it is unavailable when we need only a couple of lines later. This will also help with the proof that maxUtil satisfies the specification, leading to the realization that proving "the correctness of a program . . . at the same time it is being synthesized" is sometimes the simplest way to go.

The usage of unsafe coercions leads to the possibility of increasing the correctness of our programs "from below". They indicate the key points we need to address to improve the implementations and prove they meet the specifications.

In summary, formulating the current specifications and implementations in constructive type theory does not result in immediate ideal correctness for our programs, which is perhaps disappointing. On the other hand, it leaves us no worse off than before, and it offers a clear path for improvement: we can better our specifications by making them more constructive, and we can refine our implementations by removing unsafe features and postulates.

4 Conclusions

The approach to increasing the correctness of scientific computing presented here requires us to formalize the current typical mixture of classical and constructive mathematics within an implementation of constructive type theory, using the kind of brute-force mechanisms that the dependently-typed programming community rightly frowns upon. Nevertheless, the possibility to use unsafe features and postulates, together with a good foreign-function interface, is essential if we want to take advantage of what we can do *now*, in our less than ideal circumstances.

The results of such a formalization are, as we have seen in the previous section, not very pretty, but they have the advantage of explicitly flagging the points where improvements can be made. We are then in a better position to move towards the Martin-Löf ideal of correctness, by replacing unsafe coercions and postulates.

In many cases, this requires a shift in the programming style we adopt. There are (at least) two ways of specifying a computation with inputs of type A and outputs of type B which have to satisfy a relation R:

1. as a member of the type

   $$\forall\,[\mathsf{a} : \mathsf{A}]\ (\exists\,[\mathsf{b} : \mathsf{B}]\ (\mathsf{R}\ \mathsf{a}\ \mathsf{b}))$$

   that is, a function which, for every input $\mathsf{a} : \mathsf{A}$ returns a pair consisting of a value $\mathsf{b} : \mathsf{B}$ together with a proof that $\mathsf{R}\ \mathsf{a}\ \mathsf{b}$. This is the approach presented in the textbook of Nordström et al. [14];
2. or as a member of the type

   $$\exists\,[\mathsf{f} : (\mathsf{A} \to \mathsf{B})]\ (\forall\,[\mathsf{a} : \mathsf{A}]\ (\mathsf{R}\ \mathsf{a}\ (\mathsf{f}\ \mathsf{a})))$$

   that is, as a pair consisting of a function $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ and a proof that for every $\mathsf{a} : \mathsf{A}$ we have $\mathsf{R}\ \mathsf{a}\ (\mathsf{f}\ \mathsf{a})$. This is the approach of the other major textbook on programming and type theory, that of Thompson [20].

The two approaches are logically equivalent (see also [21]), in the sense that a member of one can always be turned in a member of the other, but they encourage a different practice: the Nordström et al. style suggests developing the proof *within* the implementation, while the Thompson style advocates developing the proof *alongside* the implementation.

The Thompson approach fits very well the current state of affairs in scientific computing, where implementations are generally considered separately from specifications, and is therefore easier for newcomers to type theory to understand. This is the style we saw in the previous section, when we translated the Haskell function maxUtil more or less literally in Agda, and postulated that it fulfills the specification maxFinSpec. This approach is also forced on us whenever we use an external function without a type-theoretical interface.

However, this style usually leads to duplication of effort: the same kind of operations needed to implement the computation turn out to be useful for proving that it fulfills the specification. Moreover, in the absence of a powerful reflection mechanism, the proof cannot be formulated at all. This was the case in the example from the previous section, where, after the boolean test, we "lost" the information necessary to prove the type-correctness of our program and had to appeal to unsafe coercion.

Once these difficulties are encountered in practice, the newcomers quickly start to appreciate the value of developing the proof and the program in the same context. The increase in correctness is accompanied by a gradual shift from the Thompson-style to the Nordström-style. This is not so surprising, after all the observation that correct-by-construction programming is easier than separating proof and program predates Martin-Löf's type theory (e.g. Dijkstra [8]).

5 Appendix

We have summarized here the definitions of the datatypes and functions we have used in this article and which are not part of the standard Agda syntax or library. For more information on the latter, the reader is referred to the Agda wiki [1], where a wealth of material is available, starting with the ever-growing reference manual. For an introduction to the Martin-Löf type theory that Agda is based on we recommend, besides the standard textbooks referred to in Section 4, also [15]. The various video lectures and associated materials of Conor McBride, such as [13], are an excellent and entertaining introduction to programming with dependent types.

Back to the Agda definitions: these fall in three categories. The first comprises datatypes and functions designed to maximize the amount of cut-and-paste we can do between Agda and Idris programs:

```
data ⊥ : Set where

data One : Set where
  one    : One

so : Bool → Set
so true = One
so false = ⊥
```

```
data _∧_ (A : Set) (B : Set) : Set where
  _,_ : A → B → A ∧ B
fst : {A B : Set} → A ∧ B → A
fst (a, b) = a
snd : {A B : Set} → A ∧ B → B
snd (a, b) = b
```

The second category consists of definitions and syntax declarations meant to increase the similarity of Agda and standard logic notations:

```
data ∃ (A : Set) (B : A → Set) : Set where
  _,_ : (a : A) → (b : B a) → ∃ A B
syntax ∃ A (λ x → B) = ∃ [x : A] B
Π : (A : Set) → (B : A → Set) → Set
Π A B = (a : A) → B a
syntax Π A (λ x → B) = ∀ [x : A] B
```

The third category is that of function definitions that are meant to facilitate the usage of unsafe features and the literal translation of Haskell programs. The only such function we have used here is

```
coerce' : {A B : Set} → A ≡ B → A → B
coerce' refl a = a
coerce : {A B : Set} → A → B
coerce = coerce' trustMe
```

References

1. Agda wiki page. `http://wiki.portal.chalmers.se/agda/`.
2. Formalisation of Mathematics. `http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ForMath`.
3. GEM-E3 Website. `http://www.gem-e3.net/`.
4. ReMIND-R. `http://www.pik-potsdam.de/research/sustainable-solutions/models/remind`.
5. R. E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
6. D. P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 2nd edition, 2000.
7. E. Brady and K. Hammond. Correct-by-construction concurrency: Using dependent types to verify implementations of effectful resource usage protocol. *Fundamenta Informaticae*, 102:145–176, 2010.
8. E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3):174–186, 1968.
9. P. Evensen and M. Märdin. An extensible and scalable agent-based simulation of barter economics. Master's Thesis 2009/04a. Chalmers Univ. of Techn. & U. of Gothenburg., 2009.

10. H. Gintis. The emergence of a price system from decentralized bilateral exchange. *The B.E. Journal of Theoretical Economics*, 6(1):13, 2006.
11. V. Kreinovich. Designing, understanding, and analyzing unconventional computation: The important role of logic and constructive mathematics. *Applied Mathematical Sciences*, 6(13-16):645–649, 2012.
12. P. Martin-Löf. Constructive mathematics and computer programming. *Philosophical Transactions of the Royal Society of London*, 312(1522):501–518, 1984.
13. C. McBride. Dependently typed programming. `http://www.cs.uoregon.edu/Research/summerschool/summer10/curriculum.html`.
14. B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory.* Oxford University Press, 1990.
15. B. Nordström, K. Petersson, and J. Smith. Martin-Löf's type theory. *Handbook of logic in computer science*, 5:1–37, 2000.
16. B. Nordström and J. Smith. Propositions and specifications of programs in Martin-Löf's type theory. *BIT Numerical Mathematics*, 24:288–301, 1984.
17. P. Suppes. *Introduction to logic.* University series in undergraduate mathematics. D. Van Nostrand Co., 1957.
18. P. Suppes. *Introduction to Logic.* Dover Books on Mathematics Series. Dover, 1999. Reprint of [17].
19. W. Swierstra. A Hoare logic for the state monad. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 440–451, Berlin, Heidelberg, 2009. Springer-Verlag.
20. S. Thompson. *Type theory and functional programming.* Addison-Wesley, 1991.
21. S. Thompson. Are subsets necessary in martin-löf type theory? In J. Myers, J.Paul and M. O'Donnell, editors, *Constructivity in Computer Science*, volume 613 of *Lecture Notes in Computer Science*, pages 46–57. Springer, 1992.
22. R. Turner. *Computable Models.* Springer, 2009.
23. H. R. Varian. *Microeconomic analysis.* Norton New York, 1992.
24. K. Velupillai. *Computable Economics: The Arne Ryde Memorial Lectures.* Oxford University Press, 2000.
25. K. V. Velupillai. Algorithmic foundations of computable general equilibrium theory. *Applied Mathematics and Computation*, 179:360–369, 2006.
26. K. V. Velupillai. Taming the incomputable, reconstructing the nonconstructive and deciding the undecidable in mathematical economics. *New Mathematics and Natural Computation (NMNC)*, 8(01):5–51, 2012.
27. L. Walras. *Elements of Pure Economics: Or the Theory of Social Wealth.* Routledge Library Editions-Economics, 40. Taylor & Francis Group, 1954.