Software Development for the Working Actuary*

David Raymond Christiansen

IT University of Copenhagen
drc@itu.dk

Abstract. We present an in-progress domain-specific language for actuaries. Due to the mathematical sophistication of actuaries and the relatively high degree of formalization of the field, we conjecture that a dependently-typed functional language with special support for actuarial models will enable actuaries to develop software that is robust and understandable.

1 Introduction

The demands on insurers and actuarial software are increasing. New rules from the European Union, called *Solvency II* [6], pose new challenges that require significant changes to pension infrastructures. The Actulus project, a collaboration between the University of Copenhagen, Edlund A/S and the IT University of Copenhagen, seeks to solve this problem through a combination of actuarial science and programming language research, taking advantage of Edlund's position as a market-leading vendor of software to the life insurance and pension industry in Denmark.

Key to our approach is empowering actuaries to develop their own analysis tools. We aim to do this by developing a domain-specific language that supports actuarial models and yet is sufficiently general to express a wide variety of programs.

Actuaries are an interesting target for end-user software development for a number of reasons:

- Actuarial science is a highly-formalized field with well-understood terms and ideas.
- Actuaries are used to formal notation and mathematical thinking.
- Many actuaries write software already in the course of their day-to-day work.

This paper describes ongoing work on a domain-specific language for actuaries that will enable them to safely develop models of life insurance and pension products that can be analyzed either with a standard set of utilities or with tools that the actuaries construct themselves. Sect. 2 presents just enough of the theory in question to explain the features of the language. Sect. 3 presents our preliminary solutions. Finally, Sect. 4 discusses our plans for further developing our language.

* Work supported by the Danish Advanced Technology Foundation (*Højteknologifonden*) (017-2010-3)

2 Actuarial Theory

In this section, the basics of actuarial mathematics as relevant to Actulus are laid out. Note that the mathematical models influence the development of new products, as only products that can be effectively modeled are sold, while pressures from the market can lead to new models. It is far beyond the scope of this paper to provide a proper introduction to actuarial mathematics, but Promislow's textbook [5] contains a discussion of the multi-state, continuous-time models that are used.

An insurance or pension product is a collection of *states* that one or more people can inhabit, combined with repeating payments due during sojourn in these states and lump-sum payments due on transition from one state to another.

Both pricing of products and determination of solvency rely on calculating what is known as the *reserves* of a product. The reserve is the present value of the expected future payments, both benefits and premiums.

Two simple products will be used as running examples in this discussion: the *pure endowment* and the *temporary life annuity*.

Pure endowment If the policy holder is alive at some time t, then he or she receives a payment.

Temporary life annuity The policy holder receives a continuous payment from time 0 until some time n or until his or her death, whichever occurs first.

More complicated products can depend on multiple related lives (such as those of a married couple and their children), track more events (such as disability), and have more complicated specifications of payments.

In Actulus, products are modeled within continuous-time Markov processes. Continuous-time Markov processes are similar to the perhaps more familiar discrete-time Markov processes, except instead of defining state-transition probabilities as a function of time, we define state-transition *intensities*. Integrating a transition intensity over an interval yields the probability of transition in that interval.

Sometimes, we also want to make use of semi-Markov models, in which transition intensities can depend on duration of sojourn in the present state. The optimal treatment of these models in the context of Actulus is ongoing work.

In addition to the Markov model representing the events in the life or lives of the insured, the payments mandated by the product must be specified. These are customarily specified in differential form, so that integrating them over an interval gives the payments due in that interval. Additionally, products can specify lump-sump payments due at state transitions.

Combining the Markov model with the payment streams and lump-sum payments allows us to construct *Thiele's differential equations*, a system of equations whose solution represents the *statewise reserves* for the product. The statewise reserves are the contribution of each state in the Markov model to the whole of the reserves, and the current reserve is the statewise reserve for the present state of the insured.

3 Actulus Modeling Language (AML)

According to the experience of Edlund A/S in the Danish insurance software industry, the present activities of the Danish life insurance and pension industry are largely centered around a compendium of well-understood products with standardized mathematical models, called *G82*. This rules out broad classes of interesting products and it can lead to the use of models that ignore important features of the actual products being modeled if these stray too far from *G82*.

When a fitting model has been chosen or developed, actuaries send the model to professional software developers. The professional programmers then implement the necessary calculations. Many actuaries are familiar to an extent with programming, especially using languages such as R or Matlab, but they are not typically skilled in modern programming languages and software development practices. Because of this disconnect, the correspondence between the model and the resulting code is not necessarily readily apparent.

By using a numerical differential equation solver combined with a domain-specific language for describing products and the calculations involving them, the Actulus project aims to make insurance calculation both more accurate and more understandable. Actulus consists of the following components: a high-performance numerical solver for differential equations called the *calculation kernel*, a language for declaratively defining the products described in Sect. 2 called the *product language* or AML-P, and a language for describing calculations called the *calculation language* or AML-C. While it is extremely important in practice for Actulus, the calculation kernel is not the focus of this paper. Instead, we describe AML, as it is the primary interface between the user and the system.

There have been a number of examples of DSLs for financial applications in the literature, dating at least back to Risla in 1995 [1]. One particularly interesting line of research is a number of Haskell-based combinator libraries for defining derivatives products that are based on work by Peyton Jones, Eber and Seward [4]. A paper by Mogensen that describes the use of a novel type system for a financial DSL [2] is also highly relevant.

While other pension administration and calculation systems present forms or tables to the actuary as a primary interaction model, Actulus will present the user with a programming language. While work on AML is not yet complete, the broad outlines of its structure are apparent. We believe that making it easy for actuaries to program their systems themselves will allow them to receive many of the benefits of modern programming languages, such as modular constructions, code re-use, and the safety provided by an advanced type system, when constructing their models.

AML is a dependently-typed total functional language with special support in the type system for checking specific properties of insurance products. These insurance products are defined using a special syntax that is designed to be particularly readable for domain experts.

Because the dedicated product syntax adheres closely to the actuarial theory described in Sect. 2, we expect that actuaries who are not skilled software developers will be able to read and write these descriptions with little additional

training. This will be empirically evaluated when an implementation is ready. Additionally, we expect that the presence of a mathematically-inspired programming language will allow users to incrementally build on their knowledge of the modeling language. However, the product modeling notation alone is at least as expressive as existing forms-and-tables interfaces. All products from *G82* can be expressed in AML-P.

The AML product language consists of three primary constructions: state models, products, and risk models. Products represent a collection of (conditional) payments. Risk models describe the transition intensities. State models simply ensure that products and compatible risk models are used together. Both of our example products in Sect. 2 mention only two states: `alive` and `dead`. We call this state model `LifeDeath`:

```
statemodel LifeDeath where
  states = alive | dead
  transitions = alive → dead
```

When we define a product, we must specify the state model within which it is defined. Syntactically, this resembles a type ascription.

```
product PureEndowment(expiry : TimePoint) : LifeDeath where
  obligations = at t pay ¤1 when (t = expiry) provided (alive)

product TempLifeAnn(expiry : TimePoint) : LifeDeath where
  obligations = at t pay ¤1 per year provided (alive and t < expiry)
```

In the above DSL, **at** binds the variable `t` representing the current point in time, the operator ¤ constructs currency, **per year** constructs a constant-rate payment stream from an amount of money, **when** constructs a payment stream that delivers a lump sum at a particular point in time, and **provided** makes a payment stream conditional on a particular set of states or Boolean conditions.

The other half of our model, the transition intensities, is described by a risk model. An example risk model for `LifeDeath` follows:

```
riskmodel Mortality(p : Person) : LifeDeath where
  intensities = alive → dead by gm(p)
```

In the above, `gm` is a library function that takes a representation of a person (presumably some sort of record structure with fields for age, sex, and so forth) as its argument, returning a mortality intensity according to the industry-standard Gompertz-Makeham formula. As described in Sect. 2, a transition intensity is the continuous-time analog of a transition probability in discrete-time Markov chains.

The *statewise reserves* are the contributions that each state makes to the overall reserve. We can compute it for these two products by calling the appropriate library function:

```
statewiseReserves(PureEndowment(TimePoint(2035, 1, 1)), Mortality)
```

There are a few important details in the call to `statewiseReserves`. First, the fact that PureEndowment and Mortality are defined according to the same state model - `LifeDeath` - can be statically checked. Second, we don't need to write that the state model in question is in fact `LifeDeath`, because it can be inferred from the other two arguments.

One of the main features that distinguishes our work on AML from other domain-specific languages is that it is a *total* language with *dependent types*. A total language is one in which every function returns a result for every type-correct input. In other words, infinite loops are impossible and no pattern-match may miss a case. By construction, this prevents a number of errors, and it results in a language that more closely matches mathematical notions of functions. For example, we don't need to worry about the potential for an infinite recursion in a transition intensity.

Perhaps the most daring design choice in AML is the inclusion of full dependent types. Dependent types are types that can be abstracted over values, and not just other types as in traditional functional programming. In many ways, dependent types represent a quite radical departure from traditional functional programming, and a full introduction to them is beyond the scope of this paper. Oury and Swierstra [3] offer an accessible introduction to the expressive capabilities of dependent types.

Dependent types allow the type system to be much more precise. For example, a list type can encode the length of the list, and the `head` function can then, in its type, require that the list is non-empty.

For perhaps a more relevant example of the kinds of invariants that can be enforced through dependent types, consider discounting of money. Discounting is the process of using information about interest rates to compute the value of an amount of money at some other time. Because one euro in 2003 is worth a different amount than one euro in 2013, we can't simply add currency. Instead, we must *discount* the value, converting from one time to another.

For the sake of simplicity, assume that the interest rate is some constant r through the entire period. In that case, we discount from t_0 to t_1 by multiplying the value at t_0 by $e^{r(t_1-t_0)}$. In AML, we can force this discounting by adding a point in time to our currency type. We then require that these time points be equal in the types of arguments to our addition function.

We begin by defining a data type `Money` which is *indexed by* a point in time. Two instances of `Money` only have the same type if they are indexed by the same point in time, just as a Java `ArrayList<String>` has a different type than `ArrayList<File>`. Note that the types of `a` and `b` refer to the value of the parameter `t` – this is a feature of a dependent type system. The compiler will reject calls to `add` for which it cannot prove that `a` and `b` have the same time index.

```
type Money(time : TimePoint) : Type where
  Amount(x: Real) : Money(time)

function add(t : TimePoint, a : Money(t), b : Money(t)) : Money(t) where
  add(t, Amount(x), Amount(y)) = Amount(x + y)
```

We now have the ability to represent currency at a particular point in time. We can define a discounting function according to the above formula as follows:

```
function discount(t0 : TimePoint, t1 : TimePoint,
                  interest : Real, m0 : Money(t0)): Money(t1) where
  discount(TimePoint(t0'), TimePoint(t1'), r, Amount(x0)) =
    Amount(exp(r * (t1' - t0')) * x0)
```

By controlling the scope of the `Amount` constructor, the library author can prevent explicit pattern-matching on the value as was done in the definition of `discount`. An important potential source of errors is eliminated entirely through judicious use of dependent types in three short, readable definitions.

4 Future work

The ongoing development of AML has two primary aspects:

- We want to ensure that AML is on a strong theoretical footing, with a sound type system. Users should not be able to circumvent the typechecker, whether through malice or by accident.
- We need to ensure that AML is actually usable by actuaries.

We plan to achieve the first goal through a combination of explaining the unique features of AML in terms of well-understood systems and through a mixture of machine-checked and manual mathematical reasoning. The second goal, however, requires empirical evaluation. Implementation work on an AML interpreter has begun. We will soon be in a position to begin testing the language and associated tools with actuaries.

References

1. B. R. T. Arnold, Arie van Deursen, and Martijn Res. Algebraic specification of a language for describing financial products. Technical report, Eindhoven University of Technology, 1995.
2. Torben Mogensen. Linear Types for Cashflow Reengineering. In Manfred Broy and Alexandre Zamulin, editors, *Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, chapter 2, pages 823–845. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2003.
3. Nicolas Oury and Wouter Swierstra. The power of pi. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 39–50, New York, NY, USA, 2008. ACM.
4. Simon Peyton-Jones, Jean M. Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). *SIGPLAN Not.*, 35(9):280–292, September 2000.
5. S. David Promislow. *Fundamentals of Actuarial Mathematics*. Wiley, second edition, 2011.
6. Directive 2009/138/EC of the European Parliament and of the council.