Software components as invariant-typed arrows
(Keynote Talk)

Luis S. Barbosa

HASLab - High Assurance Software Laboratory,
INESC TEC & Universidade do Minho, Portugal
lsb@di.uminho.pt

Abstract. Invariants are constraints on software components which restrict their behavior in some desirable way, but whose maintenance entails some kind of proof obligation discharge. Such constraints may act not only over the input and output domains, as in a purely functional setting, but also over the underlying state space, as in the case of reactive components. This talk introduces an approach for reasoning about invariants which is both compositional and calculational: compositional because it is based on rules which break the complexity of such proof obligations across the structures involved; calculational because such rules are derived thanks to an algebra of invariants encoded in the language of binary relations. A main tool of this approach is the pointfree transform of the predicate calculus, which opens the possibility of changing the underlying mathematical space so as to enable agile algebraic calculation. The development of a theory of invariant preservation requires a broad, but uniform view of computational processes embodied in software components able to take into account data persistence and continued interaction. Such is the plan for this talk: we first introduce such processes as *arrows*, and then invariants as their *types*.

1 Components as arrows

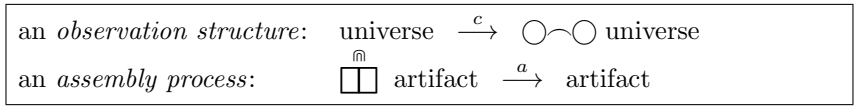
Probably the most elementary model of a computational process is that of a *function* $f : I \longrightarrow O$, which specifies a transformation rule between two structures I and O. In a (metaphorical) sense, this may be dubbed as the 'engineer's view' of reality: *here is a recipe to build gnus from gnats*. Often, however, reality is not so simple. For example, one may know how to produce 'gnus' from 'gnats' but not in all cases. This is expressed by observing the output of f in a more refined context: O is replaced by $O + \mathbf{1}$ and f is said to be a *partial* function. In other situations one may recognise that there is some *context* information about 'gnats' that, for some reason, should be hidden from input. It may be the case that such information is huge to be give as a parameter to f, or shared by other functions as well. It might also be the case that building gnus would eventually modify the environment, thus influencing latter production of more 'gnus'. For U a denotation of such context information, the signature of f becomes $f : I \longrightarrow (O \times U)^U$. In both cases f can be typed as $f : I \longrightarrow T\,O$,

for $T = \mathsf{Id} + \mathbf{1}$ and $T = (\mathsf{Id} \times U)^U$, respectively, where, intuitively, T is a type transformer providing a *shape* for the output of f. Technically, T is a *functor* which, to facilitate composition and manipulation of such functions, is often required to be a *monad*. In this way, the 'universe' in which $f : I \longrightarrow T\,O$ lives and is reasoned about is the *Kleisli category* for T. In fact, monads in functional programming offer a general technique to smoothly incorporate, and delimit, 'computational effects' of this kind without compromising the purely functional semantics of such languages, in particular, referential transparency.

A function computed within a context is often referred to as 'state-based', in the sense the word 'state' has in automata theory — the memory which both constrains and is constrained by the execution of actions. In fact, the 'nature' of $f : I \longrightarrow (O \times U)^U$ as a 'state-based function' is made more explicit by rewriting its signature as $f : U \longrightarrow (O \times U)^I$

This, in turn, may suggest an alternative model for computations, which (again in a metaphorical sense) one may dub as the 'natural scientist's view'. Instead of a recipe to build 'gnus' from 'gnats', the simple awareness that *there exist gnus and gnats and that their evolution can be observed*. That *observation* may entail some form of *interference* is well known, even from Physics, and thus the underlying notion of computation is not necessarily a passive one.

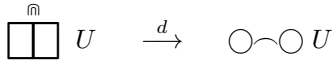
The able 'natural scientist' will equip herself with the right 'lens' — that is, a tool to observe with, which necessarily entails a particular shape for observation. Similarly, the engineer will resort to a 'tool box' emphasizing the possibility of at least some (essentially finite) things being not only observed, but actually *built*. In summary,



Assembly processes are specified in a similar (but dual) way to *observation structures*. Note that in the picture 'artifact' has replaced 'universe', to stress that one is now dealing with 'culture' (as opposed to 'nature') and, what is far more relevant, that the arrow has been *reversed*. Formally, both 'lenses' and 'toolboxes' are functors. And, therefore, an *observation structure* is a ◯⌒◯-coalgebra, and an *assembly process* is a ⊓⊔-algebra.

Algebras and coalgebras for a functor [13] provide abstract models of essentially construction (or *data*-oriented) and observation (or *behaviour*-oriented) computational processes, respectively. Construction *compatibility* and *indistinguishability* under observation emerge as the basic notions of equivalence which, moreover, are characterized in a way which is parametric on the particular 'toolbox' or 'lens' used, respectively. Algebraic compatibility and bisimilarity *acquire a shape*, which is the source of abstraction such models are proud of. Moreover, it is well known that, if 'toolboxs' or 'lens' are 'smooth enough', there exist *canonical* representations of all 'artifacts' or 'behaviours into an initial (respectively, final) algebra (respectively, coalgebra).

Both *assembly* and *observation* processes, as discussed above, can be modeled by functions, or more generally, by arrows in a suitable category, between the *universes-of-interest*. Both aspects can be combined in a single arrow



formally known as a *dialgebra*. Initially defined in [14], their theory was developed in [15, 1] and later by [16] in the style of *universal algebra*. In Computer Science, dialgebras were firstly used in [7] to deal with data types in a purely categorical way and more recently in [11], as a generalization of both algebras and coalgebras. In [12], they are used to specify systems whose states may have an algebraic structure, i.e., as models of evolving algebras [6].

Dialgebras ($d : \mathsf{F}\,U \longrightarrow \mathsf{G}\,U$) generalize many interesting computational structures, among which algebras ($a : \mathsf{F}\,U \longrightarrow U$) and coalgebras ($c : U \longrightarrow \mathsf{G}\,U$) as the simplest instantiations. A basic example is provided by transition systems with specified initial states. If the transition shape is given by G, functor $Id + \mathbf{1}$ introduces initial states as constants. This makes possible, for example, to introduce initial states on models of automata, as in $d : Q + 1 \longrightarrow Q^{In} \times 2$. Another example are components whose services may have a non deterministic output. If functor F captures an algebraic signature, $d : \mathsf{F}\,U \longrightarrow \mathcal{P}(U)$ caters for non deterministic outcomes.

2 Invariants as types

If dialgebras provide a very general model for computational processes regarded as arrows between the *universes-of-interest*, one has also to be precise on what such 'universes' really are. A key observation is that, along their lifetime, computer systems are expected to maintain a certain number of properties on which depend their consistency and data integrity. On the other hand, they are subject to the permanent stress of ever changing business rules, which materialise into (either static or dynamic) properties of the underlying code.

Both integrity constraints and domain business rules are examples of *invariant* properties. The word 'invariant' captures the idea that such desirable properties are to be maintained *invariant*, that is, unharmed across all transactions which are embodied in the system's functionality.

Invariants are ubiquitous in systems design. Actually, they take several forms and are defined not only over the input and output domains, as in a purely functional setting, but also over the underlying state space, as in imperative programming or reactive systems design. Software evolution and reconfiguration, on the other hand, entails the need for invariant checking whenever running code is upgraded or even dynamically reconfigured. While testing is the most widely used technique for such purpose, it is highly costly and does not ensure correctness. Ideally, one should be able to *formally verify* that the new invariants are enforced without running the (new) code at all.

This calls for a general theory of invariant preservation upon which one could base such an extended static checking mechanism. This talk sums up a number of steps towards such a theory which is both

- *compositional*: based on rules which break the complexity of the relevant proof obligations across the structures involved
- *calculational*: amenable to agile algebraic manipulation

Our starting point is the explicit use of relational techniques, a body of knowledge often referred to as the *algebra of programming* [5]. In particular an invariant $P \subseteq X$ is represented as a binary relation $y\ \Phi_P\ x\ \equiv\ y = x\ \wedge\ x \in P$, which is called *coreflexive* because it is a fragment of the identity relation, *i.e.*, $\Phi_P \subseteq id$. Notice this is one of the standard ways of encoding a set as a binary relation. Since predicates and coreflexives are in one to one correspondence, we will use uppercase Greek letters to denote such coreflexives and will refer to them as 'invariants' with no further explanation.

Then, we resort to such relations to model types for arrows representing computational processes. Actually, if one regards invariants as *types*, the computational processes they type are *arrows*:

$$\mathsf{F}\,\Phi_P \xrightarrow{\ d\ } \mathsf{G}\,\Phi_P \tag{1}$$

where $\mathsf{F}\,\Phi$ and $\mathsf{G}\,\Phi$ represent invariant Φ placed in a *context* abstracted by functors F and G, in the sense discussed above.

Typing computational processes (modelled as dialgebras) by invariants encodes a *proof obligation*. Actually the meaning of arrow (1) is

$$d \cdot \mathsf{F}\,\Phi_P\ \subseteq\ \mathsf{G}\,\Phi_P \cdot d \tag{2}$$

which is computed as the relational counterpart to the following first-order formula $\langle \forall\ u\ ::\ u \in \mathsf{F}(P)\ \Rightarrow\ d(u) \in \mathsf{G}(P) \rangle$.

The intuition behind this move is that a dialgebra typed by a predicate is a structure for which such a predicate is to be maintained along its evolution. We will show how this can generalised in the context of a category whose objects are predicates and arrows encode proof obligations, cf,

- for general functions: $\Phi \xrightarrow{\ f\ } \Psi$
- for *reactive processes* modelled as dialgebras $\mathsf{F}\Phi \xrightarrow{\ d\ } \mathsf{G}\Phi$
- for *imperative programs*: $\Phi_{pre} \xrightarrow{\ R\ } \Phi_{post}$ corresponding to *Hoare triples* $\{post\}R\{pre\}$. This requires a generalization of the invariant calculus to *relations*, to capture the calculus of weakest pre-conditions.

In each case, a calculus of invariants' proof obligation discharge is developed, generalising our previous work. References [2, 8, 3, 9] provide a roadmap through our research on (coalgebraic) calculi for components-as-arrows. Most results on typing such arrows by predicates first appeared in [4], with further developments in [10].

Acknowlegments.

Long time collaboration with J. N. Oliveira (Minho), Alexandra Silva (Nijmegen) and Manuel A. Martins (Aveiro), is deeply acknowledged. This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - the Portuguese Foundation for Science and Technology within project **FCOMP-01-0124-FEDER-010047**.

References

1. Jirí Adámek. Limits and colimits in generalized algebraic categories. *Czechoslovak Mathematical Journal*, 26:55–64, 1976.
2. L. S. Barbosa. Towards a Calculus of State-based Software Components. *Journal of Universal Computer Science*, 9(8):891–909, August 2003.
3. L. S. Barbosa and J. N. Oliveira. Transposing partial components: an exercise on coalgebraic refinement. *Theor. Comp. Sci.*, 365(1-2):2–22, 2006.
4. L. S. Barbosa, J. N. Oliveira, and A. M. Silva. Calculating invariants as coreflexive bisimulations. In J. Meseguer and G. Rosu, editors, *Proc. 12th Inter. Conf. on Algebraic Methodology and Software Technology, AMAST*, pages 83–99. Springer Lect. Notes Comp. Sci. (5140), 2008.
5. R. Bird and O. Moor. *The Algebra of Programming.* Series in Computer Science. Prentice Hall, 1997.
6. E. Börger and R. Stärk. *Abstract state machines: A method for high-level system design and analysis.* Springer-Verlag, 2003.
7. Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In *Category Theory and Computer Science*, pages 140–157, 1987.
8. Sun Meng and L. S. Barbosa. Components as coalgebras: The refinement dimension. *Theor. Comp. Sci.*, 351:276–294, 2005.
9. Sun Meng and L. S. Barbosa. Towards the introduction of qos information in a component model. In S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, and C.-C. Hung, editors, *Proceedings of the 2010 ACM Symposium on Applied Computing, Sierre, Switzerland*, pages 2045–2046. ACM, 2010.
10. J. N. Oliveira. Extended static checking by calculation using the pointfree transform. In Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto, editors, *Language Engineering and Rigorous Software Development, Inter.l LerNet ALFA Summer School 2008, Piriapolis, Uruguay, Revised Tutorial Lectures*, volume 5520 of *Lecture Notes in Computer Science*, pages 195–251. Springer, 2009.
11. Erik Poll and Jan Zwanenburg. From algebras and coalgebras to dialgebras. In *CMCS '01, volume 44 of ENTCS*, pages 1–19. Elsevier, 2001.
12. Horst Reichel. Unifying adt– and evolving algebra specifications. *EATCS Bulletin*, 59:112–126, 1996.
13. J. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249:3–80, 2000.
14. V. Trnková and P. Goralcík. On products in generalized algebraic categories. *Commentationes Mathematicae Universitatis Carolinae*, 1:49–89, 1972.
15. Věra Trnková. On descriptive classification of set-functors. I. *Commentat. Math. Univ. Carol.*, 12:143–174, 1971.
16. G Voutsadakis. Universal dialgebra: Unifying universal algebra and coalgebra. *Far East Journal of Mathematical Sciences*, 44(1), 2010.