

Type Systems, Software Architecture and Testing

Stuart O. Anderson
LFCS, Department of
Computer Science
University of Edinburgh
Edinburgh EH9 3JZ, UK
+44 131 650 5191
soa@lfcs.ed.ac.uk

Daniele Compare
Dip. Di Mat. Pura ed Applicata
Universita' dell'Aquila
Via Vetoio, Localita' Coppito
L'Aquila, Italy
compare@univaq.it

ABSTRACT

Modern type systems for programming languages offer an expressive language for talking about software components and provide tools that offer thorough analysis of such components and their interconnection. We argue that many of the concerns in testing and analysis of software architectures are also the concerns in the programming language type system community but they are expressed in somewhat different terms. We exemplify this using small examples drawn from Standard ML and Extended ML. We also point out some recent developments in type systems that may be of relevance to the software architecture community.

KEYWORDS

type systems, Standard ML, polymorphism, module systems, interface specifications, test case generation, adequacy of test cases

Over the past two decades the programming language design community has developed increasingly sophisticated type systems. These offer well-packaged and powerful tools for the analysis of programming systems. These systems are usually decidable and include features such as polymorphism (of various kinds), higher order objects, and more recently subtyping and inheritance. Type systems generally fall into two kinds: those that check type annotations are correct and those that infer the type of objects in the system. Cardelli [1] provides an excellent survey of much of the work in the area.

The position I hold is that modern type systems can be of great value in analysing and testing software architectures. Here, my argument for this position is to provide a couple of simple examples of architecture descriptions in a language with a modern type system (Standard ML (SML) [3]). My position is *not* that SML is the ideal Architecture Description Language (ADL). My argument is that work on ADLs could be enriched by work on Type Systems and that some information derived from type systems can help in the testing of architectures. The Type System literature has not been completely ignored, see [2], but I feel it could be given greater attention.

I give two short examples of different styles of type system used in SML to give some idea of the range of information derivable. Because space is short the examples are small but I believe they could be scaled up quite easily. In both examples I choose to use pipe and filter style architecture, other styles can be accommodated.

SML consists of two almost completely independently defined parts. The *core* is a call by value functional language with some imperative features. The typing mechanism in the core is type inference which means that the user of the system can omit typing information completely and the system will infer the most general type for any expression. The *module* system of SML is intended as a notation for describing how to build ML systems. It is explicitly typed. The user of the system must include type annotations and the system checks they are satisfied by the program.

1 Core SML: Polymorphic Type Inference

In the core of SML types are built up from constant types: e.g. `bool`, `int` and `unit` (the one element type whose value only value is `()`, the empty tuple); type variables: written `'a`, `texttt'b` and type constructors that generate new types from old: e.g. `a*b` is the type of pairs, and `a -> b` is the type of functions from `a` to `b`. The set of type constructors is not fixed, there is a basic set that can be extended as necessary.

In this section we will use the example shown in Figure 1. This is a simple pipe and filter architecture where the filters are represented by circles, pipes by the arcs and the component in the square is not a filter but rather something that generates some kind of summary of its input on its output.

Implementing such an architecture in SML is fairly straightforward¹. The `stream` type constructor is introduced to model infinitely proceeding streams and then the skeleton components are defined.

```
datatype 'a stream =
```

¹This is *not* intended to be an efficient implementation. It contains a number of significant inefficiencies.

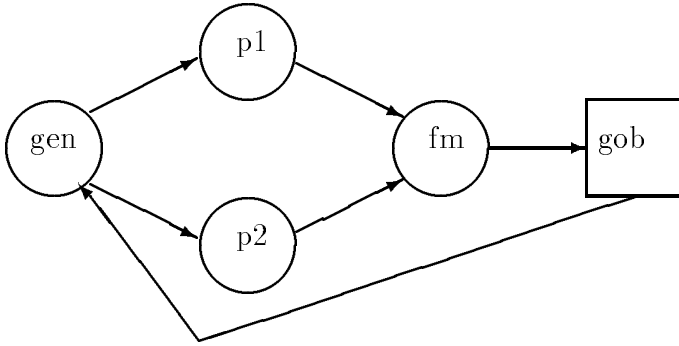


Figure 1: A Simple Architecture

```

NEXT of (unit -> 'a * 'a stream)

fun filter f (NEXT t) =
  NEXT(fn ()=>let val (v,s) = t()
             in
               (f v, filter f s)
             end)

fun merge f (NEXT t) (NEXT t') =
  NEXT(fn () => let val (v,s) = t()
                  val (v',s') = t'()
                  in
                    (f v v', merge f s s')
                  end)

fun gobble g s =
  NEXT(fn () => let val (v,s) = g s
                 in
                   (v,gobble g s)
                 end)

fun sys gen p1 p2 fm gob init =
  let
    fun f i =
      let val s0 =
          filter gen
            (NEXT(fn () => (i,f i)))
          val s1 = filter p1 s0
          val s2 = filter p2 s0
          val s3 = merge fm s1 s2
        in
          gobble gob s3
        end
  in
    f init
  end

```

The function `sys` describes the architecture. Notice that the functions associated with each of the nodes in the architecture are parameters to the function so that this

is the generic form of the architecture that can be specialised by supplying specific functions that implement the component operations at the nodes in the diagram.

The type inference system of SML responds with the following typing for the system. The types allocated are the most general possible in the sense that any other typing of the term is obtainable by substituting some type for a type variable in the type. Here is the system response:

```

datatype 'a stream =
  NEXT of unit -> ('a * 'a stream)
val NEXT : (unit -> ('a * 'a stream)) ->
  'a stream

val filter : ('a -> 'b) -> 'a stream ->
  'b stream = fn
val gobble : ('a -> ('b * 'b stream)) ->
  'a -> 'b stream = fn
val merge : ('a -> 'b -> 'c) -> 'a stream ->
  'b stream -> 'c stream = fn
val sys : ('a -> 'b) ->
  ('b -> 'c) ->
  ('b -> 'd) ->
  ('c -> 'd -> 'e) ->
  ('e stream -> ('a * 'a stream)) ->
  'a -> 'a stream = fn

```

Looking at the type inferred for `sys` we can see that the typechecker has correctly inferred the various argument/result agreements that are required in the architecture. For example, the output of the `gob` node must agree with the input to the `gen` node.

If we choose to specialise the architecture by providing some interpretation for the functions at the node then the type systems ensure the specialisations. For example, if we choose to implement `fm` as pairing and `gob` as the identity then we require that the input to `gen` is a pair:

```

fun specsys gen p1 p2 init =
  let fun fm a b = (a,b)
      fun gob (NEXT t) = t()
    in
      sys gen p1 p2 fm gob init
    end

```

The type inferred for the specialised system carries this information:

```

val specsys :
  (('a * 'b) -> 'c) ->
  ('c -> 'a) ->
  ('c -> 'b) ->

```

```
('a * 'b) -> ('a * 'b) stream = fn
```

It is quite possible to make incorrect refinements of the architecture. For example, if we make `p1` the identity in the previous system then we end up with a contradiction which is detected as an occurs check problem during unification.

```
fun syserr p2 init =
  let fun p1 a = a
        fun gen a = a
        fun fm a b = (a,b)
        fun gob (NEXT t) = t()
    in
      sys gen p1 p2 fm gob init
    end
```

Function applied to argument of wrong type

```
Near: sys gen p1 p2 fm gob
Required argument type:
  ('b * 'a) stream ->
    ('b * ('b * 'a) stream)
Actual argument type:
  ('b * 'a) stream ->
    (('b * 'a) * ('b * 'a) stream)
Circular type results from unifying
  'b
and
  ('b * 'a)
```

We know that types catch a significant number of programming errors and that polymorphic types capture parametric generality. If architectures are generic in this sense it seems that languages with parametric polymorphism and type inference could capture some aspects of generic architectures.

1.1 Implications for Testing

The main motivation for this discussion of parametric polymorphism is in the context of architecture testing. If we believe that architectures can be generic, then those architectures may have polymorphic types that capture generic operations. If a parameter to a function is allocated a type variable by the typechecker then that data is not subjected to inspection by the function. The appearance of such polymorphism means we have no need to test polymorphic values with an extensive test set. If the function works for one value of a polymorphic argument it will work for all possible values (see [6]).

For example, the function:

```
fun systest init =
  let fun p1 a = a
```

```
    fun p2 a = a
    fun gen a = a
    fun fm a b = (a,b)
    fun gob (NEXT t) =
      let val ((a,_),t) = t()
    in (a,t) end
    in
      sys gen p1 p2 fm gob init
    end
```

has type `'a -> 'a stream`. To test the function we do not need to consider more than one input because the behaviour is identical for all possible values. Wherever we see type variables we can test with the unit type. This can bring a significant reduction in the effort required to test some architecture.

1.2 Extensions to the Type System

The basic ML typesystem has been extended in a variety of ways. For example, the treatment of equality in SML has led to an extension of the type system to keep track of those variables that are tested for equality. More recently Wadler [7] has developed a linear type system that detects those values that are used only once in the course of a computation. It may be that knowing facts like these can help in testing programs.

2 SML Modules: Typed Interfaces

In the SML module system we manipulate *structures*. These are collections of constructs in the language. The contents of structures are described by *signatures* (signatures are rather like types for structures) and *functors* are typed maps from structures to structures. This approach leads to a style of system construction where all the module interfaces are well documented by signatures (that documentation can be augmented with axioms in the Extended ML specification language[5]). The example below illustrates the flavour of the approach. For a fuller account of the use of Modules as an architecture implementation language see [4].

```
datatype ('a,'b) Filter =
  WAITIN of ('a -> ('a,'b)Filter) |
  WAITOUT of ('b * ('a,'b)Filter) |
  READY of (unit -> ('a,'b)Filter)

signature FILTER =
sig
  type Messin
  type Messout
  type UsrMess
```

```
  val filter: (Messin,Messout)Filter
end;
```

```

functor Pipe(structure A: FILTER
             structure B: FILTER
             sharing type A.Messout = B.Messin)
             : FILTER =
struct
  type Messin = A.Messin
  type Messout = B.Messout

  val filter =
    let fun pipe(READY f,x) = pipe(f(),x)
        | pipe(x,READY g) = pipe(x,g())
        | pipe(WAITOUT(mo,fa),WAITIN g) =
            pipe(fa,g mo)
        | pipe(x,WAITOUT(mo,fb)) =
            WAITOUT(mo,
                    READY(fn () => pipe(x,fb)))
        | pipe(WAITIN f,fb) =
            WAITIN(fn mi => pipe(f mi,fb))
    in
      pipe(A.filter,B.filter)
    end
end;

```

2.1 A Small Experiment

Recently we have been carrying out some small experiments into the utility of the typed interfaces in the Module system as a basis for integration testing. Using a type-driven test case generator and working from an outline proof of correctness of an Extended ML specification we considered the adequacy of the test cases generated using only the interface specification.

So far we do not have enough data to make any conclusive claims but it does appear that using the interface to generate test cases does uncover errors in components that have been in widespread use. The reasons seem fairly clear, the approach of using the signature to generate test cases results in cases that are outside the normal operating range of the component and thus these cases have never been exercised.

2.2 Further Developments

The study of type systems for modules is an active area of research. For an up-to-date survey and some interesting new results see [8].

Higher Order Functors Here we allow functors to take functors as parameters, the result is a more expressive system with the possibility of building connector combinators in a convenient way.

First Class Structures This would allow a mixing of the core and module level. This could provide a sound basis for dynamic reconfiguration of systems.

Ambient/Global Computation As we move to an environment where computational services are offered on a global scale we need to find ways to trust and test systems whose functionality we do not completely control and about which we may have less than complete information for an introduction see [9].

3 Conclusions

I believe that the interaction between modern type systems, software architectures and testing could be fruitful. I hope this paper provides some justification for that belief.

REFERENCES

- [1] L. Cardelli. **Type Systems** Available at: <http://research.microsoft.com/research/cambridge/luca/Papers/TypeSystems.pdf>
- [2] R. J. Allen. **A Formal Approach to Software Architecture** CMU-CS-97-144
- [3] R. Milner, M. Tofte and R. Harper. **The Definition of Standard ML** MIT Press, 1989.
- [4] E. Biagioni, R. Harper and P. Lee. **Implementing Software Architectures in Standard ML** *Proc. ACM ICSE 17*
- [5] D. Sannella. *Formal Program Development in Extended ML for the Working Programmer*, LFCS Report ECS-LFCS-89-102, 1989.
- [6] P. Wadler. **Theorems for free!** 4th International Conference on Functional Programming and Computer Architecture, London, September 1989.
- [7] P. Wadler. **Linear types can change the world!** In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland, Amsterdam, 1990.
- [8] C. Russo. **Types for Modules** LFCS Thesis 1998.
- [9] L. Cardelli. **Global Computation** Available at: <http://research.microsoft.com/research/cambridge/luca/Papers/GlobalComputation.html>