SML Prototypes from Z Specifications

Greg Michaelson

Department of Computing and Electrical Engineering
Heriot-Watt University, Riccarton, EH14 4AS, UK
greg@cee.hw.ac.uk

Abstract. The wider uptake of Declarative Meta-Programming may be constrained by a perceived separation of formal notations and declarative techniques from practical system construction. A pedagogic approach to demonstrating the relevance of formal and declarative techniques, through the prototyping Z of specifications in Standard ML, is discussed.

1 Introduction

In principle, Declarative Meta-Programming (DMP) offers a promising realisation of formal techniques for software development through tools written in declarative languages. DMP is based on manipulation of program code, often specified as compositional abstract syntax tree transformations in some formal notation. Such notations tend to have a close correspondence to declarative languages; hardly surprising given that formalisms for describing language semantics have common roots with declarative languages in number theoretic predicate calculus, set theory, recursive function theory and λ calculus. Thus, declarative languages are a good first choice for practical experimentation with DMP.

However, the wider uptake of DMP may be compromised by the unwarranted aura of academic obscurity and mathematicity that attends its foundations. In particular, formal approaches and declarative programming are often taught in isolated, specialist modules in Computer Science programmes. Thus, students see little relevance to mainstream software engineering and, as Computing professionals, are unlikely to promote them once in employment.

The origins of these techniques' isolation may lie in the Platonist views of many of their strongest supporters. Such views gained currency in the 1970s, in debates about the relative merits of operational and denotational semantics (DS), summarised in [Mic93]. Members of the Oxford Programming Research Group who developed DS, argued for its primacy as static descriptions of ideal abstract entities rather than as potentially executable specifications of languages. In particular, they asserted that an operational reading of a DS imputed unnecessary, non-unique, implementation details [SS71,Sco72,Sto77].

More recently, Hayes and Jones [HJ89] argued against the executability of specifications in general. Like the DS proponents, they asserted that specification notations encompass infinite domains and non-computable constructs, and underdetermine equivalent implementations. They also said that executable notations require implementation properties that are irrelevant from a formal perspective.

Ironically, people working with formal specifications often wish to explore empirically their properties. Thus, specifications may be "animated" using symbolic techniques like abstract interpretation. If concrete values are substituted for symbols, and animation terminates, then the final result might be interpreted as an output from the execution of an equivalent program.

In contrast, there has been considerable research into refinement techniques for deriving software from specifications. However, the target has usually been unimplemented Djikstra/Hoare/Gries guarded command notations for which proof techniques are easily elaborated [Kal90,Mor90]. Furthermore, there is a dearth of tools to aid refinement to standard languages.

The significant exception to these gloomy observations has been the development of Abrial's B-Method [Abr96], which integrates specification, proof, refinement and implementation through the unifying concept of abstract machines. Refinement is based on successive steps to replace abstract with more concrete constructs. Each step must be justified through a linking invariant, which relates the initial and refined states. The invariant also provides assurances about the properties of concrete constructs.

2 Prototyping with specifications

Since 1992, we have been teaching a module on formal specification in Z [Spi89] to 2nd year BSc Computer Science students. In a recent attempt to increase relevance, we sought to develop a linked module where the students would construct programs from Z specifications.

An initial candidate was the B-Method, which certainly offers good tool support for proof and animation. However, the B notation is far closer to Z than to a standard programming language, the use of which we deemed important for emphasising the significance of formal specification for practical software development. Thus, we decided instead to trade B's rigour for a more informal approach based on the direct translation of Z to light weight prototypes in a standard language. We also thought that this would enable relatively quick experimentation with the implementation implications of specifications,

In choosing a target language, we sought a close correspondence to Z. Z, based on set theoretic predicate calculus, has the Church-Rosser property of evaluation order independence, is strongly typed and enables parametric polymorphism through generic schemas. This suggested the use of a declarative language as the target.

We chose Standard ML (SML) [MTH97], a strict, strongly type, parametric polymorphic language. SML is technically an imperative language with a pure functional subset. We thought that SML might eventually enable evolutionary prototyping, through the systematic replacement of pure functional with imperative constructs. Furthermore, there is good tool support for proving properties of SML programs. Stepney [Ste93] discusses the use of Prolog for implementing a compiler from DS specified in Z. Diller [Dil90] briefly discusses the alternatives of Miranda and Prolog for animating Z.

In the late 1980's, Murray [Mur89] explored the construction of a SML tool set for Z, making substantial use of the SML functor mechanism. While this demonstrated the feasibility of prototyping Z in SML, Murray did not elaborate a detailed methodology.

3 From Z to SML

Our translation from Z to SML is based on the simple correspondences between Z and SML types shown in Figure 1.

Z	SML
logical constants	`bool`
$\mathbb{N}$ and $\mathbb{Z}$	`int`
real numbers	`real`
$Free ::= Option1 \mid Option2$	`datatype Free = Option1	Option2`
arbitrary constant	`string`
$X \times Y$	`X * Y`
$\mathbb{P}\, S$	`S list`
$X \leftrightarrow Y$	`(X * Y) list`

Fig. 1. Z/SML type correspondences.

Rather than requiring the full translation of Z schema to problem specific SML, like Murray we provide SML libraries of polymorphic higher order functions to support the standard Z set, relation and function operations. See Figure 3.

Then, to construct a prototype:

- SML types and type aliases are used to define Z types;
- the state schema is translated to a boolean assertion function;
- for each schema, the pre-condition is translated to a boolean function and the post-condition is translated to a function returning a tuple of all state variables;
- the total schemas are combined in a single function of associated free type values;
- the total scheme function is called by a wrapper function that tests the state schema assertion as an invariant and raises exceptions appropriately;
- test cases are devised to satisfy and fail all pre-conditions, and the invariant assertion.

For example, consider the schema shown in Figure 3 for withdrawing a quantity of an item from a warehouse[Lig01].

Z	SML
$x \in S$	`member x S`
	`member : ''a -> ''a list -> bool`
$\forall x : S \bullet p(x)$	`all p S`
	`all : ('a -> bool) -> 'a list -> bool`
$\exists x : S \bullet p(x)$	`exists p S`
	`exists : ('a -> bool) -> 'a list -> bool`
$S1 \subseteq S2$	`subset S1 S2`
	`subset : ''a list -> ''a list -> bool`
$S1 = S2$	`equals S1 S2`
	`equals : ''a list -> ''a list -> bool`
$S1 \cup S2$	`union S1 S2`
	`union : ''a list -> ''a list -> ''a list`
$S1 \cap S2$	`intersect S1 S2`
	`intersect : ''a list -> ''a list -> ''a list`
$S1 \setminus S2$	`diff S1 S2`
	`diff : ''a list -> ''a list -> ''a list`
$a \mapsto b$	`(a,b) : A * B`
$A \leftrightarrow B$	`(A * B) list`
$\mathrm{dom}\, S$	`dom S`
	`dom : ('a * 'b) list -> 'a list`
$\mathrm{ran}\, S$	`ran S`
	`ran : ('a * 'b) list -> 'b list`
$R \llparenthesis S \rrparenthesis$	`image R S`
	`image : (''a * ''b) list -> ''a list -> ''b list`
R^{-1}	`inverse R`
	`inverse : ('a * 'b) list -> ('b * 'a) list`
$S \lhd R$	`domrest R S`
	`domrest : (''a * ''b) list -> ''a list -> (''a * ''b) list`
$S ⩤ R$	`domantirest R S`
	`domantirest : (''a * ''b) list -> ''a list -> (''a * ''b) list`
$R \rhd T$	`ranrest R T`
	`ranrest : (''a * ''b) list -> ''b list -> (''a * ''b) list`
$S ⩥ R$	`ranantirest R S`
	`ranantirest : (''a * ''b) list -> ''b list -> (''a * ''b) list`
$f\ x$	`fapply f x`
	`fapply : (''a * 'b) list -> ''a -> 'b`
$f \oplus g$	`over f g`
	`over : (''a * 'b) list -> (''a * 'b) list -> (''a * 'b) list`

Fig. 2. SML library functions for Z operators.

$$
\begin{array}{|l}
\hline
\textit{Withdraw} \\
\Delta \textit{Warehouse} \\
i? : \textit{ITEM} \\
qty? : \mathbb{N}_1 \\
\hline
i? \in \textit{carried} \\
\textit{level}\ \ i? \geq qty? \\
\textit{level}' = \textit{level} \oplus \{i? \mapsto (\textit{level}\ \ i? - qty?)\} \\
\textit{carried}' = \textit{carried} \\
\hline
\end{array}
$$

Fig. 3. Z schema for withdrawing quantity of item from warehouse

Here, $\textit{level} : \textit{ITEM} \mapsto \mathbb{N}_1$ records how many of each item are held in stock and $\textit{carried} : \mathbb{P}\,\textit{ITEM}$ records which items are held.

Types for $\textit{ITEM}$, *level* and set of $\textit{ITEM}$ are defined as shown in Figure 3. Note that $\mathbb{N}_1$ is represented as `int`.

```
type ITEM = string;
type LEVEL = (ITEM * int) list;
type CARRIED = ITEM list;
```

Fig. 4. SML representations for Z warehouse types

The schema may be translated as shown in Figure 5.

```
- fun preWithdraw (carried:CARRIED,i:ITEM) =
       member i carried andalso
       fapply level i >= qty;
> val preWithdraw = fn : CARRIED * ITEM -> bool

- fun doWithdraw (level:LEVEL,carried:CARRIED,i:ITEM,qty:int) =
       (over level [(i,fapply level i - qty)],
        carried)
> val doWithdraw = fn : LEVEL * CARRIED * ITEM * int -> LEVEL * CARRIED
```

Fig. 5. SML for withdrawing quantity of item from warehouse.

4 Discussion

The new module was taught for the first time in 2002, to 56 students. Module assessment included a practical exercise based on a simple air traffic control example [Lig01], specified as 3 total schema built from 8 auxiliary schema. The example is actually incompletely specified: most of the students spotted and corrected this while constructing their prototypes. Almost all of the students successfully completed the exercise. At the end of the module, implementation in Java from an SML prototype was briefly discussed.

Students were asked to comment on how this approach compared with their previous preference, which most nominated as procedural implementation in Java. Overall, the students were positive about prototyping Z in SML, often commenting that it seemed to reduce errors and development time, and increased assurance that the software met the specification. Many also commented that prototyping had greatly increased their command of Z. Some suggested that the approach might be better suited to much larger problems. However, a minority remained unhappy, preferring the direct Java hack.

We are pleased that this approach appears to have made formal specification and declarative programming more accessible to our students. However, the approach, while formally motivated is clearly informal. It might be placed on more solid foundations by proving the SML libraries correct relative to standard set theoretic definitions of the Z operators, and then investigating the use of a theorem prover to try and establish the correctness of SML prototypes relative to Z specifications. It would also be interesting to look at building tools to assist and automate the approach.

Acknowledgments

I would like to thank Ken Robinson, Julian Richardson and Hunter Davis for fruitful discussion. I would also like to thank my students for constructive feedback on the 2nd year *Prototyping from Specification* module.

Further details may be found in:
`http://www.macs.hw.ac.uk/~greg/F22HW3`.

References

[Abr96] J-R. Abrial. *The B Book.* CUP, 1996.

[Dil90] A. Diller. *Z: an Introduction to Formal Methods.* Wiley, 1990.

[HJ89] I. J. Hayes and C. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–338, 1989.

[Kal90] A. Kaldewaij. *Programming: the derivation of algorithms.* Prentice-Hall, 1990.

[Lig01] D. Lightfoot. *Formal Specification Using Z.* Palgrave, 2001.

[Mic93] G. J. Michaelson. *Interpreter Prototypes from Formal Language Definitions.* PhD thesis, Heriot-Watt University, 1993.

[Mor90] C. Morgan. *Programming from specifications.* Prentice-Hall, 1990.

[MTH97] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[Mur89] J. Murray. ML Prototypes from Z Specifications. Master's thesis, Heriot-Watt University, 1989.

[Sco72] D. Scott. Lattice theory, data types and semantics. In R. Rustin, editor, *Courant Computer Science Symposium 2: Formal semantics of programming languages*, pages 65–106, 1972.

[Spi89] J. M. Spivey. *The Z Notation.* Prentice Hall, 1989.

[SS71] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. Technical Report PRG-19, Computer Laboratory Programming Reserach Group, University of Oxford, August 1971.

[Ste93] S. Stepney. *High Integrity Compilation.* Prentice-Hall, 1993.

[Sto77] J. E. Stoy. *Denotational semantics: the Scott-Strachey approach to programming language theory.* MIT Press, 1977.