Common Interests between Specware and MMT

Florian Rabe

November 7, 2014

Abstract

MMT is a framework for representing declarative languages such as logics, type theories, set theories, etc.. It achieves a high level of generality by systematically avoiding a commitment to a particular syntax or semantics. That makes MMT particularly suitable for integrating different languages, libraries, and systems.

Specware is an environment for the design, development and automated synthesis of scalable, correct-by-construction software.

This note discusses the approach, language, and system of Specware from an MMT perspective and identifies potential synergies in the joint development of a future system.

It is meant to be read by people familiar with Specware (see `http://www.specware.org/`) and with the general introduction to MMT available at `https://svn.kwarc.info/repos/MMT/doc/introduction/mmt.pdf`.

1 The Specware Language as an Interface Logic

The language underlying Specware can be seen as a foundation in the sense of MMT. Consequently, it can be represented in a logical framework inside MMT, and Specware theories can be exported to MMT. This would require the same steps as for any other foundation and is worthwhile for the same reasons. Some features are difficult to represent, in particular the details of inductive and record types, but most aspects are straightforward to represent in MMT.

However, **Specware is different from most foundations** considered in MMT so far:

- Firstly, it uses a rich **type system** that goes beyond most foundations. In particular, Specware's partial functions, comprehension and quotient types, and description operator are unusual: They focus on intuitive expressivity close to common concepts in software and mathematics at the price of making the type system undecidable.
- Secondly, the language uses **no formal semantics**. In particular, it does not give typing and proof rules that would define well-formedness and theoremhood. Instead, it relies on an informal description that yields an implicit but intuitively clear set theoretical semantics. This is somewhat to PVS [ORS92].
- Thirdly, the language includes propositions but **no proofs**. Instead, proofs are omitted or provided as proof scripts for external theorem provers.

These differences stem from the motivation of supporting software engineering, which leads to different design priorities.

But incidentally, these differences also make Specware very **close to the interface logics that are needed** to integrate large libraries of formal logical knowledge. These interface logics are

feature-rich but ultimately relatively inexpressive languages that focus on conveniently specifying data structures and operations without committing to a proof system.

2 The Specware System

The Specware systems implements only parts of the semantics directly, in particular, disambiguation and the decidable parts of type checking. **Specware generates proof obligations** to handle the undecidable aspects of type checking[1] and all aspects regarding proving theorems. The latter includes also the decidable fragments of the logic, e.g., quantification over a finite type.

The proof obligations can be **exported to external provers** (currently only Isabelle [Pau94]) for verification. To integrate Specware with external verification, Specware files may provide proofs in the prover's native syntax, which are carried through in the export.

This design choice was likely made out of necessity – building and maintaining a Specware-specific theorem prover is simply too expensive. But it can also be seen as an asset – the design space remains unconstrained by the limitations of a theorem prover and the choice of theorem prover remains flexible.

In comparison to Mmt, we note:

- Mmt allows users to write specifications in the interface logic of their choice. The Specware language should be one of these interface logics. However, the complexity of logics that can be defined in logical frameworks does not reach the one of Specware yet.
- Like Specware, Mmt does not insist on validity being decidable and instead collects the proof obligations induced by a declaration.
- Contrary to Specware, Mmt includes a native theorem prover (very basic at the moment but gradually becoming stronger in the long run). Mmt does not export proof obligations to external provers yet, but this is a goal of future work.

The Specware community has built multiple exports to theorem provers (ACL2 [KMM00], Coq [Coq14]), but their maturity and maintenance states are unclear. It is reasonable to assume that it is expensive to obtain and maintain multiple translations. Therefore, Specware would benefit from a central mediator for distributing proof obligations to different provers and integrating their results.

3 UniFormal

In the past, mediators that bring together producers and consumers of proof obligations have been built with great success for the special case of first-order logic. Here the TPTP interchange language [SS98] has had enormous effects in standardizing, comparing, and integrating theorem provers. However, designing a corresponding system for feature-rich logics is much harder because the producer and consumer of the proof obligations will typically use very different and often incompatible foundations.

UniFormal is a hypothetical system envisioned as the central mediator by Lambert Meertens. The Mmt language and system provide a very promising starting point for building UniFormal.

[1] It is not entirely clear to the author at this point whether all proof obligations induced by the type system are generated.

Given a central repository of definitions of interface logics (such as the LATIN library) and interface theories, systems are able to produce proof obligations as a tuple of:

- an interface logic L expressed as an MMT theory,
- an L-theory T that describes the current context, possibly importing some interface theories,
- a proof obligation F stated in T.

UniFormal can then use L to identify theorem provers P that are able to understand T and F, translate T and F into the input syntax of P, and forward the proof obligation to P.

Because interface logics are developed modularly, L will usually just list the needed features, e.g., the feature of type comprehension is only needed if T and F actually use it. This has several advantages:

- Proof obligations can be stated in the simplest possible interface logic, thus maximizing the set of available provers.
- Translations of interface languages into theorem provers can be developed feature-wise. That makes it easy to build translations gradually, and partial translations can be used immediately when applicable.
- MMT theory morphisms in the interface library can be used to translate features into each other. Thus, UniFormal can eliminate features that are not supported by a theorem prover in terms of supported features.

The use of a central library of interface logics permits relegating the task of importing proof obligations to the prover developers. This has advantages for both systems:

- Producers of proof obligations do not have to develop and maintain (often brittle) connections to theorem provers. Instead, they only have to maintain a single, relatively easy export to the interface logics.
- Consumers of proof obligations can import them directly from UniFormal's standardized syntax, thus by-passing the tedious generation and parsing of concrete syntax. This would also allow every theorem prover to expose their services to multiple systems.

References

[Coq14] Coq Development Team. The Coq Proof Assistant: Reference Manual. Technical report, INRIA, 2014.

[KMM00] M. Kaufmann, P. Manolios, and J Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752. Springer, 1992.

[Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[SS98] G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

A The Specware Language for MMT Insiders

This is a very quick overview of the abstract syntax of Specware (excluding the theory level) from the perspective of MMT.

Concepts Specware uses 3 concepts:

- types,
- typed terms with internal propositions (terms of built-in type `Boolean`),
- proofs asserting a proposition (All proofs are opaque and given in the syntax of external provers.)

Declarations Specware allows declaring constants for types, terms, and assertions, all of which may be defined. It supports 2 extension patterns for named inductive and quotient types (see below).

Specware uses shallow polymorphism: All declarations may take type parameters. Arguments of term constants are inferred or provided by annotating an expression with its type.

Minor aspects:

- The type of a term constant must be provided.
- The type and the definiens of a type or term constant may be given in separate declarations.
- Assertion declarations with definiens are annotated as either theorem or conjecture, depending on whether the proof has been checked externally.
- Term constants may carry a notation, which is infix-left or infix-right and a precedence.
- Overloading is not supported, but the unqualified use of qualified identifiers is resolved based on the available type information.

Expressions The Specware uses the following type constructors, each grouped with their introductory and elimination form:

- Type of Booleans. Constants for true, false, binary (infix) constants for equivalence, implication, disjunction, conjuction, polymorphic equality and inequality, and flexary quantifiers for universal, existential, unique existential. Elimination via a constant for if-then-else.
- Simple function types. Constants for λ-abstraction and application. Inductive functions may directly use patterns.
- Simple flexary product types. Constants for tupling and projection. The polymorphic selection function (`project`n) is an expression by itself.
- Record types (Single-field records are not allowed; the empty record and the empty product are identified.) Constants for record formation and selection. The polymorphic selection function (`project`$name$) is an expression by itself. A special constant $<<$ is used for merging two record values, whose types agree on the common fields; the record values do not have to agree on the common fields, and the right operand's values are used.
- Predicate subtypes (called comprehension types or type restrictions depending on notation). No constants for introduction or elimination because implicit coercions are used based on the expected type.

The 2 extension patterns introduce the following term constructors:

- Named inductive types. Constants for introduction are introduced by the declaration. Elimination is done by patterned case distinction; patterns may be guarded and non-exhaustive.
- Named quotient types ($Q = T/q$ for $q : T * T \to$ `Boolean`). Constants for introduction (`quotient`$[Q] : T \to Q$) and elimination (`choose`$[Q] : \{f : T \to R | \ldots\} \to (Q \to R)$).

Further expression constructors that are not associated with a type constructor:

- Expressions annotated with their type.
- A flexary let binders. Recursion is allowed.
- A polymorphic description operator (called unique-solution). The well-formedness obligation is assumed.
- Sequential expressions $(A; B)$ can be used to write function calls that may have side-effects in models.

Built-in types with literals are

- natural numbers,
- characters,
- strings.

For some types introduced in the Specware library, special notations are built-in:

- the usual notations for a unary type operator for lists,
- notations for monadic unary type operators as defined by a built-in specification.