A simple refinement language for Casl

Till Mossakowski[1], Don Sannella[2], and Andrzej Tarlecki[3]

[1] BISS, Department of Computer Science, University of Bremen
[2] LFCS, School of Informatics, University of Edinburgh, Edinburgh, UK
[3] Institute of Informatics, Warsaw University and Institute of Computer Science, PAS, Warsaw, Poland

Abstract

The standard development paradigm of algebraic specification [1] postulates that the development begins with a formal *requirement specification* (extracted from a software project's informal requirements) that fixes only expected properties but ideally says nothing about implementation issues; this is to be followed by a number of *refinement* steps that fix more and more details of the design, so that one finally arrives at what is often termed the *design specification*. The last refinement step then results in an actual *implementation* in a programming language.

In this work, we extend the specification language Casl [6] with a simple refinement language and also address the passage to implementations in functional programming languages (SML and Haskell). The refinement language is much simpler that that originally proposed in draft versions of [6].

Casl structured specifications can be seen as specifications of non-parameterized program units, whereas Casl unit specifications also cover program units that are parameterized by other program units. Of particular importance are *monomorphic* unit specifications, that is unit specifications whose result specification is a monomorphic extension of the argument specifications. Up to isomorphism, these specify just one parameterized program. Of course, in order to be able to actually translate such a unit specification to a program in a programming language PL, further PL-specific restrictions on monomorphic unit specifications have to be imposed: for functional languages, we require that all sorts are given as free types, and functions are defined recursively in a way that termination is provable. In this case, an easy translation to a functional program is possible [5], see [3] for a translation to OCAML. Using free extensions, it is also possible to capture partial recursive functions, see [5, 3]. Moreover, with Haskell (and its type class Eq) as target language, also generated types with explicitly given equality can be used.

Refinement now basically means that unit specifications are refined into unit specifications, until a monomorphic unit specification is reached. We provide a simple notation for this

refine USP_1 **to** USP_2

The semantics is that each parameterized unit satisfying USP_2 also satisfies USP_1.

CASL architectural specifications [2] allow to introduce branches into such linear refinement sequences; they describe (via unit terms) how several (possibly parameterized) units can be combined into a new one. A unit $U : USP_1$ in an architectural specification ASP can be refined with

refine U **from** ASP **to** USP_2

with the same semantics as above. Finally, since CASL allows a coercion from architectural specifications to unit specifications, we allow architectural specifications to occur directly as targets of refinements. (Note that this also allows for simulating refinements along signature morphisms: the reduction against the signature morphism can be just put into the result unit term of the architectural specification).

Given a programming language PL, a *complete refinement tree* for PL has a unit specification as its root, and architectural or unit specifications as its inner nodes, such that

- each unit specification has a unique child, to which it is refined,
- each architectural specification has one child for each of its units, such that the specification of the unit is refined to the child, and
- each leaf is a monomorphic unit specification satisfying the PL-specific restrictions.

Any such refinement tree can then be translated into a collection of PL-modules jointly implementing the root specification of the tree, provided that PL has a module system that is rich enough to capture CASL unit terms. For ML and Haskell, this looks as follows:

CASL	ML	Haskell
non-parameterized unit	structure	module
parameterized unit	functor	multi-parameter type class in a module
monomorphic unit specification with free types and recursive definitions	structure with free types and recursive definitions	module with free types and recursive definitions
unit application	functor application	type class instantiation
unit amalgamation	combination of structures	combination of modules
unit hiding	restriction to subsignature	hiding
unit renaming	redefinition	redefinition
architectural specification	functor depending on other functors	module depending on other modules

The possibility of the following alternative design is still under discussion: instead of letting monomorphic unit specifications be the end-point of a development, let the end-point be architectural specifications without any declared units. This makes sense if the language for architectural unit terms is rich enough to express all the needed programming language-specific constructors. In order to achieve this richness, the language has to be extend e.g. by constructors for free extensions and institution-specific inductive schemata for the definition of functions. This has the disadvantage of an institution-specific language for architectural unit terms, but the advantage that from the beginning, constructors are expressed directly rather than indirectly, up to isomorphism, via a specification. On the other hand, monomorphic unit specifications satisfying the syntactic PL-specific restrictions for a programming language PL typically will have a semantics giving unique models (and not only models unique up to isomorphism) as well. This might lead to a reconciliation of the two approaches.

References

[1] E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, *Algebraic foundations of systems specification*, Springer, 1999.

[2] Michel Bidoit, Donald Sannella, and Andrzej Tarlecki, *Architectural specifications in CASL*, Formal Aspects of Computing **13** (2002), 252–273.

[3] Thibaud Brunet, *Génération automatique de code à partir de spécifications formelles*, Master's thesis, Université de Poitiers, 2003.

[4] CoFI, *The Common Framework Initiative for algebraic specification and development, electronic archives*, Notes and Documents accessible from `http://www.cofi.info/`.

[5] Till Mossakowski, *Two "functional programming" sublanguages of CASL*, Note L-9, in [4], March 1998.

[6] Peter D. Mosses (ed.), *CASL reference manual*, Lecture Notes in Computer Science, vol. 2960, Springer, 2004, To appear.