Specifying and developing parameterised programs*
CFI Language Design Study Notes

Don Sannella and Andrzej Tarlecki
The Common Framework Initiative

March 8, 1996

The enterprise of formal specification and development is relevant to parameterised programs, with exactly the same motivation as in the case of ordinary non-parameterised programs. An additional advantage this brings is that it enables the overall shape of a development tree to be given without the need to supply the parameterised programs involved in each of the steps. The provision of a parameterised program that fits into each step can then be regarded as a separate task, perhaps involving further refinement and decomposition. This is one of the ideas underlying the Extended ML methodology for formal development of Standard ML programs from specifications [ST 89,San 91].

For any two signatures Σ and Σ', we can regard $\Sigma{\rightarrow}\Sigma'$ as a new kind of signature. Then $Alg(\Sigma{\rightarrow}\Sigma')$ is the set of all *parametric* $(\Sigma{\rightarrow}\Sigma')$*-algebras*, that is functions $F : Alg(\Sigma) \rightarrow Alg(\Sigma')$. Just as ordinary programs are modelled as algebras, parameterised programs are modelled as parametric algebras. (We generalise this further to multi-argument and higher-order parametric algebras below.)

To specify a parameterised program, we give its input and output interfaces. The specification $SP{\rightarrow}SP'$ describes the class of parametric $(Sig(SP){\rightarrow}Sig(SP'))$-algebras $F : Alg(Sig(SP)) \rightarrow Alg(Sig(SP'))$ such that $F(A) \in [\![SP']\!]$ for all $A \in [\![SP]\!]$. Said another way, $Sig(SP{\rightarrow}SP') = Sig(SP){\rightarrow}Sig(SP')$ and $[\![SP{\rightarrow}SP']\!] = \{F \in Alg(Sig(SP){\rightarrow}Sig(SP')) \mid F([\![SP]\!]) \subseteq [\![SP']\!]\}$. The statement that κ is a realization of $SP{\rightarrow}SP'$ is thus equivalent to the correctness of the constructor implementation $SP' \underset{\kappa}{\rightsquigarrow} SP$. (Thus "constructor" and "parametric algebra" are names for the same concept: a function mapping algebras to algebras.) The specification $SP{\rightarrow}SP'$ is *not* a so-called *parameterised specification*; it is a non-parameterised specification of a parameterised program. See [SST 92] for a discussion of this distinction.

Example 1
Let `SORT` be a specification of sorting lists of integers and let `INTLIST` be a specification of lists of integers with an order relation on integers. `INTLIST`$\rightarrow$`SORT` specifies a parameterised program which, given an implementation of `INTLIST`, delivers an implementation of `SORT`. A realization of this specification is the functor:

```
functor K(X:INTLIST):SORT =
   struct
      open X
      fun insert(x,nil) = [x]
        | insert(x,y::l) = if le(x,y) then insert(x,y::l) = x::y::l
                           else insert(x,y::l) = y::insert(x,l)
      fun sort(nil) = nil
```

*This note is a short version of Section 8 of [ST 95]. See that paper for more examples.

```
        | sort(x::l) = insert(x,sort(l))
    end
```

That is, $[\![\mathtt{K}]\!] \in [\![\mathtt{INTLIST}{\rightarrow}\mathtt{SORT}]\!]$. $\square$

The definition of refinement of specifications applies without modification to specifications of parameterised programs:

$$SP_1{\rightarrow}SP'_1 \rightsquigarrow SP_2{\rightarrow}SP'_2 \quad \textbf{iff} \quad [\![SP_2{\rightarrow}SP'_2]\!] \subseteq [\![SP_1{\rightarrow}SP'_1]\!]$$

which presupposes that $Sig(SP_1{\rightarrow}SP'_1) = Sig(SP_2{\rightarrow}SP'_2)$, i.e. $Sig(SP_1) = Sig(SP_2)$ and $Sig(SP'_1) = Sig(SP'_2)$.

Constructor implementations may be similarly generalized to deal with specifications of parameterised programs. However, the parameterised programs used to define the constructors involved in such implementations are then higher-order, i.e. they take parameterised programs as arguments and return parameterised programs as results. Higher-order functors are not available in Standard ML as defined in [MTH 90], but their semantics and implementation is a topic of current active research [MT 94].

The definitions involved in dealing with parameterised programs and their specifications extend to the higher-order case in a natural way [SST 92]. The set of generalized signatures is defined to be the least set containing ordinary signatures and such that if $\Sigma_1, \ldots, \Sigma_n$ $(n \geq 0)$ and Σ are generalized signatures, then $\langle\Sigma_1, \ldots, \Sigma_n\rangle{\rightarrow}\Sigma$ is a generalized signature as well; if $n = 1$ we omit the brackets. $Alg(\langle\Sigma_1, \ldots, \Sigma_n\rangle{\rightarrow}\Sigma)$ is the set of all functions $F : Alg(\Sigma_1) \times \cdots \times Alg(\Sigma_n) \rightarrow Alg(\Sigma)$. As before, to specify a higher-order parameterised program, we give its input interfaces and output interface, which now may themselves be specifications of (higher-order) parameterised programs. The definitions are exactly the same as those given above. (Alternatively, multi-argument parameterised programs could be presented in their "curried" form denoting parametric algebras in $Alg(\Sigma_1{\rightarrow}(\Sigma_2{\rightarrow}\cdots(\Sigma_n{\rightarrow}\Sigma)\cdots))$.) The concept of constructor implementation carries over without modification to the case of higher-order parameterised programs and their specifications.

It is possible to restrict attention to constructors of a particularly simple form, since any constructor implementation $SP \underset{\kappa}{\rightsquigarrow} \langle SP_1, \ldots, SP_n\rangle$ may be replaced by the decomposition $SP \underset{app}{\rightsquigarrow} \langle SP_\kappa, SP_1, \ldots, SP_n\rangle$, where $SP_\kappa = \langle SP_1, \ldots, SP_n\rangle{\rightarrow}SP$ and app is the higher-order parameterised program such that $app(F, A_1, \ldots, A_n) = F(A_1, \ldots, A_n)$ and where the parameterised program κ is then provided as the realization of SP_κ. A decomposition like $SP \underset{app}{\rightsquigarrow} \langle SP_\kappa, SP_1, \ldots, SP_n\rangle$ embodies the decision to implement SP in terms of realizations of $SP_1, \ldots, SP_n$, leaving the decision of how these are used to produce a realization of SP as a *separate* development task, specified by SP_κ. This brings a bottom-up flavour into our principally top-down view of the development process.

The above presentation uses a particularly simple form of specification of parameterised programs, where the output interface does not depend on the particular realization of the input interfaces. This is not sufficient when more complex examples are considered. The necessary extra flexibility is gained by replacing the specification $\langle SP_1, \ldots, SP_n\rangle{\rightarrow}SP$ by the generalized (dependent) product $\Pi\langle X_1{:}SP_1, \ldots, X_n{:}SP_n\rangle.SP[X_1, \ldots, X_n]$. See [SST 92] for details of this and other technicalities omitted here.

Example 1 (continued)

Specifications like $\mathtt{INTLIST}\rightarrow\mathtt{SORT}$ do not capture the intention that their realizations, when given an argument X realizing `INTLIST`, should produce a realization of `SORT` that *extends* X. In particular, a realization of $\mathtt{INTLIST}\rightarrow\mathtt{SORT}$ might ignore the order relation in its argument and produce a realization of `SORT` containing a completely different order relation, together with a sorting function that is correct with respect to this new order relation rather than the one supplied in the argument. This problem can be solved by use of the following dependent product specification:

$$\Pi X{:}\mathtt{INTLIST}.\ \textbf{enrich}\ \mathtt{SORT}\ \textbf{by axioms}\ \forall x,y{:}\mathtt{int}.\ le(x,y)=X.le(x,y)$$

In case types are involved, this is the issue of *sharing* in Standard ML and the use of so-called *sharing constraints* as in Standard ML and Extended ML is one way of expressing the required dependency. □

References

[MT 94] D. MacQueen and M. Tofte. A semantics for higher-order functors. *Proc. 5th European Symp. on Programming*, Edinburgh. Springer LNCS 788, 409–423 (1994).

[MTH 90] R. Milner, M. Tofte and R. Harper. *The Definition of Standard ML*. MIT Press (1990).

[San 91] D. Sannella. Formal program development in Extended ML for the working programmer. *Proc. 3rd BCS/FACS Workshop on Refinement*, Hursley Park. Springer Workshops in Computing, 99–130 (1991).
Available in `http://www.dcs.ed.ac.uk/~dts/pub/bcs.ps`

[SST 92] D. Sannella, S. Sokołowski and A. Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. *Acta Informatica* 29:689–736 (1992).
Available in `http://www.dcs.ed.ac.uk/~dts/pub/param.ps`

[ST 89] D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. *Proc. 3rd Joint Conf. on Theory and Practice of Software Development*, Barcelona. Springer LNCS 352, 375–389 (1989).
Available in `http://www.dcs.ed.ac.uk/~dts/pub/eml-methodology.ps`

[ST 95] D. Sannella and A. Tarlecki. Model-theoretic foundations for program development: basic concepts and motivation. Report 791, Institute of Computer Science, Polish Academy of Sciences (1995).
Available in `http://www.dcs.ed.ac.uk/~dts/pub/mtf.ps`