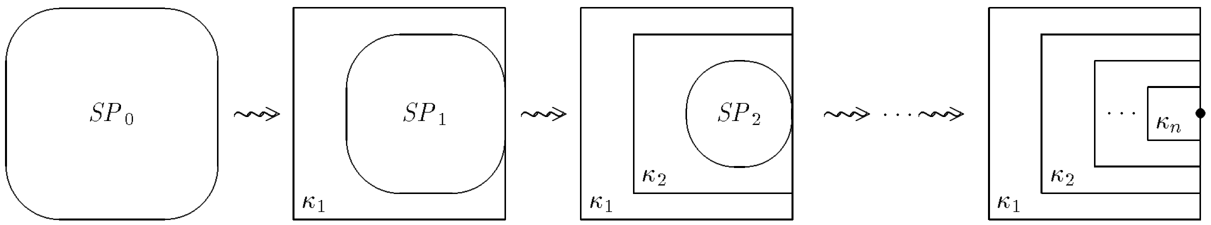
Constructor implementations*
CFI Language Design Study Notes

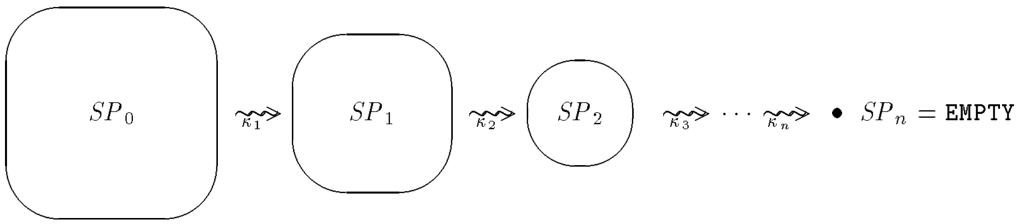
Don Sannella and Andrzej Tarlecki
The Common Framework Initiative

March 8, 1996

The simple notion of specification refinement, defined as $SP \rightsquigarrow SP'$ iff $Sig(SP) = Sig(SP')$ and $[\![SP']\!] \subseteq [\![SP]\!]$, is mathematically elegant and powerful enough (in the context of a sufficiently rich specification language) to handle all concrete examples of interest. However, it is not very convenient to use in practice. During the process of developing a program, the successive specifications incorporate more and more details arising from successive design decisions. Thereby, some parts become fully determined, and remain unchanged as a part of the specification until the final program is obtained.



It is more convenient to avoid such clutter by separating the finished parts from the specification, putting them aside, and proceeding with the development of the unresolved parts only.



It is important for the finished parts $\kappa_1, \ldots, \kappa_n$ to be independent of the particular choice of realization for what is left: they should act as constructions extending any realization of the unresolved part to a realization of what is being refined.

Each κ_i above amounts to a so-called *parameterised program* [Gog 84] with input interface SP_i and output interface SP_{i-1}. Given a program P that is a correct realization of SP_i, the parameterised program κ_i may be instantiated to yield a program $\kappa_i(P)$ that realizes SP_{i-1}. A programming language that supports stepwise development in the style suggested here needs to provide syntax and modularisation facilities for defining parameterised programs and their instantiations. For example, in the Standard ML programming language parameterised programs are called *functors*, and instantiation amounts to *functor application*. The features that the programming language provides for writing the actual code are immaterial, provided that the type system matches that of the specification language and the notion of algebra used in the semantics of the specification language is appropriate for modelling programs. Once the development is finally finished (that is, when nothing is left unresolved, as above) we can successively instantiate the parameterised programs $\kappa_n, \ldots, \kappa_1$ to obtain a correct realization of the original specification SP_0.

Semantically, each parameterised program κ_i defines a function (which we will call a *constructor*) on algebras, $[\![\kappa_i]\!] : Alg(Sig(SP_i)) \to Alg(Sig(SP_{i-1}))$, and the semantics of instantiation is simply function

*This note is a short version of Section 7 of [ST 95].

application: if $[\![P]\!] \in Alg(Sig(SP_i))$, then $[\![\kappa_i(P)]\!] = [\![\kappa_i]\!]([\![P]\!])$. In practice, κ_i provides a definition of the components (carriers and operations) of a $Sig(SP_{i-1})$-algebra, given the components of a $Sig(SP_i)$-algebra.

The above considerations motivate the notion of *constructor implementation* [ST 88]. We write $SP \underset{\kappa}{\rightsquigarrow} SP'$ to say that a specification SP' implements a specification SP via κ, where κ is a parameterised program denoting a constructor $[\![\kappa]\!] : Alg(Sig(SP')) \to Alg(Sig(SP))$, and define this as follows:

$$SP \underset{\kappa}{\rightsquigarrow} SP' \quad \textit{iff} \quad [\![\kappa]\!]([\![SP']\!]) \subseteq [\![SP]\!]$$

Here, $[\![\kappa]\!]([\![SP']\!])$ is the image of $[\![SP']\!]$ under $[\![\kappa]\!]$.

Example 1

Let `SORT` be a specification of sorting lists of integers, let `INS` be a specification of lists of integers with an operation to insert an integer into an ordered list, let `INTLIST` be a specification of lists of integers with an order relation on integers, and let `EMPTY` be the empty specification. Then the following are examples of constructor implementations:

$$\mathtt{SORT} \underset{\mathtt{K1}}{\rightsquigarrow} \mathtt{INS} \qquad \mathtt{INS} \underset{\mathtt{K2}}{\rightsquigarrow} \mathtt{INTLIST} \qquad \mathtt{INTLIST} \underset{\mathtt{K3}}{\rightsquigarrow} \mathtt{EMPTY}$$

where the constructors `K1`, `K2` and `K3` are given by the following SML functors:

```
functor K1(X:INS):SORT =
   struct
      open X
      fun sort(nil) = nil
        | sort(x::l) = insert(x,sort(l))
   end

functor K2(X:INTLIST):INS =
   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)
   end

functor K3(X:EMPTY):INTLIST =
   struct
      fun le(n,m:int) = n <= m
      fun head(x::_) = x
      fun tail(_::l) = l
      fun is_in(_,nil) = false
        | is_in(x,y::l) = if x=y then true else is_in(x,l)
   end
```

□

The correctness of the final outcome of the stepwise development process may be inferred from the correctness of the individual constructor implementation steps:

$$\frac{SP_0 \underset{\kappa_1}{\rightsquigarrow} SP_1 \underset{\kappa_2}{\rightsquigarrow} \cdots \underset{\kappa_n}{\rightsquigarrow} SP_n = \mathtt{EMPTY}}{[\![\kappa_1(\kappa_2(\ldots\kappa_n(\mathtt{empty})\ldots))]\!] \in [\![SP_0]\!]}$$

where `EMPTY` is the empty specification over the empty signature and `empty` is its (empty) realization, e.g. the SML structure `struct end`.

Example 1 (continued)

In our example,

```
K1(K2(K3(empty)))
```

yields a non-parameterised program (an SML structure) satisfying `SORT`. □

Suppose that parameterised programs compose, that is, for any two parameterised programs κ and κ' such that the signature of the output interface of κ coincides with the signature of the input interface of κ', there is a parameterised program $\kappa;\kappa'$ with $[\![\kappa;\kappa']\!] = [\![\kappa]\!];[\![\kappa']\!]$ (the latter semicolon stands for ordinary function composition, written in diagrammatic order). Then it is easy to see that constructor implementations (vertically) compose:

$$\frac{SP \underset{\kappa}{\leadsto} SP' \qquad SP' \underset{\kappa'}{\leadsto} SP''}{SP \underset{\kappa';\kappa}{\leadsto} SP''}$$

When developing a large program it is crucial to progressively decompose the job into smaller tasks that can be handled separately. Each task is defined by a specification, and solving a task means producing a program component that satisfies this specification. Once all tasks are solved, producing the final system is a simple matter of appropriately assembling these components.

A development step involving the decomposition of a programming task into separate subtasks is modelled using a constructor implementation with a multi-argument parameterised program (see [SST 92]):

$$SP \underset{\kappa}{\leadsto} \langle SP_1, \ldots, SP_n \rangle \quad \textit{iff} \quad [\![\kappa]\!]([\![SP_1]\!] \times \cdots \times [\![SP_n]\!]) \subseteq [\![SP]\!]$$

where $[\![\kappa]\!] : \mathit{Alg}(\mathit{Sig}(SP_1)) \times \cdots \times \mathit{Alg}(\mathit{Sig}(SP_n)) \rightarrow \mathit{Alg}(\mathit{Sig}(SP))$ is an n-argument constructor (an n-argument function on algebras) describing a way to put models of $SP_1, \ldots, SP_n$ together to construct a model of SP. Now the development process takes on a tree-like shape. This process is finished once a tree is obtained that has empty sequences (of specifications) as its leaves:

$$SP \underset{\kappa}{\leadsto} \left\{ \begin{array}{l} SP_1 \underset{\kappa_1}{\leadsto} \langle\,\rangle \\ \vdots \\ SP_n \underset{\kappa_n}{\leadsto} \left\{ \begin{array}{l} SP_{n1} \underset{\kappa_{n1}}{\leadsto} \left\{ SP_{n11} \underset{\kappa_{n11}}{\leadsto} \langle\,\rangle \right. \\ \cdots \\ SP_{nm} \underset{\kappa_{nm}}{\leadsto} \langle\,\rangle \end{array} \right. \end{array} \right.$$

Then an appropriate instantiation of the parameterised programs in the tree yields a realization of the original requirements specification. The above tree yields the program $\kappa(\kappa_1(), \ldots, \kappa_n(\kappa_{n1}(\kappa_{n11}()), \ldots, \kappa_{nm}()))$, with

$$[\![\kappa(\kappa_1(), \ldots, \kappa_n(\kappa_{n1}(\kappa_{n11}()), \ldots, \kappa_{nm}()))]\!] \in [\![SP]\!].$$

(We use the notation $\kappa(P_1, \ldots, P_n)$ for instantiation of n-ary parameterised programs, where $[\![\kappa(P_1, \ldots, P_n)]\!] = [\![\kappa]\!]([\![P_1]\!], \ldots, [\![P_n]\!])$.)

The structure of the final program is determined by the shape of the development tree, which is in turn determined by the decomposition steps. Each such step corresponds to what software engineers call a *design* specification (and what [GHW 82] call an *organizational* specification): it defines the structure of the system by specifying its components and describing how they fit together. This style of development leads to modular programs, built from fully specified, correct and reusable components.

References

[Gog 84] J. Goguen. Parameterized programming. *IEEE Trans. on Software Engineering* SE-10(5):528–543 (1984).

[GHW 82] J. Guttag, J. Horning and J. Wing. Some notes on putting formal specifications to productive use. *Science of Computer Programming* 2:53–68 (1982).

[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 88] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica* 25:233–281 (1988).
Available in `http://www.dcs.ed.ac.uk/~dts/pub/impl.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`