DECLARATIVE SPECIFICATION AND DECLARATIVE PROGRAMMING[1]

Manfred Broy
Institut für Informatik, Technische Universität München
Postfach 20 24 20, D-8 München, Germany

Abstract

A formalism for declarative specification and programming is introduced that forms a logical and methodological framework for program and system specification and construction. It combines axiomatic techniques based on logical concepts for specifying properties and the possibility to introduce names for objects. In particular it comprises within one formalism the possibilities of formulating specifications and defining algorithms. The logical formalism is based more or less on typed predicate logic. The development rules are particular proof rules together with the inference rules of predicate logic. As a special aspect we consider logical formulas which explicitly specify typed identifiers as elements of signatures.

1. Introduction

Recent years have shown a close relationship between programming language formalisms and logic. In predicative programming Hehner and Hoare (cf. [6], [5]) consider assignment-oriented programs as predicates or more precisely as strange notations for predicates. A programming language thus can be understood as providing a set of logical connectives that can also be expressed in and translated into terms of classical predicate logic. This way a programming language can be seen as a constructive notational extension of predicate logic. In his work on predicate transformers Dijkstra (cf. [3]) considers a program as a specific (algorithmic) notation for a function mapping predicates onto predicates. Also this way programming notations are directly translated into logical formalisms.

All of the mentioned approaches are based on the concept of state-oriented and assignment-oriented programming. In assignment-oriented languages the notion of state and the idea of updating a state is most important. Every statement changes the state or more precisely certain components of the state. The form of the state, however, is more or less kept static. In most cases the state is a simple mapping from the set of identifiers to a set of (data) values. Statements do not change the set of identifiers (program variables) by introducing new identifiers or deleting existing ones but just change the values associated with the identifiers. Declarations, however, do change the set of identifiers that are declared. In the following we introduce a logical specification and programming framework similar to the ones mentioned above based on a more functional programming style, the style of *declarative programming*. In this style the declaration of data elements and of functions, i.e. the introduction of identifiers (function symbols) by binding them to special functions is characteristic.

Declarative programming is typical for a number of more functional approaches to programming which are investigated in recent years in research on functional programming notations and in algebraic specifications. In declarative programming we specify computation structures consisting of data elements and functions by just giving formulas in predicate logic containing a number of free identifiers for sets, functions etc. that are declared by those formulas. These predicate logic formulas then define a number of properties for their free identifiers. If we in addition give indications which of the identifiers are considered as part of the input and which of the identifiers are considered as part of the output, then we have given a direction to the formula of predicate logic (cf. also the techniques in assignment-oriented programming for indicating input and output states in [5], [8]). Hence we are closer to specifications of algorithms with input and output or to parameterized specifications.

The work of [1] and of [2] suggests a consequently algebraic expression oriented style of programming. Programs are functions that are described by the

[1]This work was partially supported by the BMFT Project Korso

composition of basic functions using basic compositional forms. This style of programming is especially well-suited for the derivation and transformation of programs representing algorithms in some algebraic calculus. This implies that such a style is well suited for programming in the small. However, it is less well-suited for programming in the large, where large numbers of sorts (types) and functions have to be considered. Here the introduction (declaration) of families of names is a more helpful approach.

Declarative specification and declarative programming have also close relations to logic programming. More precisely, logic programming can be seen as an algorithmic instance of special forms of declarative specifications. In logic programming also just a set of logical formulas is given as axioms which contain a number of free identifiers, the identifiers for predicates and for logical variables. This way a number of declarations of predicates is obtained. In a query then it is referred to these identifiers declared for predicates. Further variables are mentioned in the queries that are considered as output of the algorithm. The input to the logic program is deduced implicitly by techniques of unification and resolution from the query expression.

In the following we give a syntactic and semantic framework for declarative programming. It can be seen as a generalization of axiomatic algebraic specification, logic programming, and functional programming towards a declarative specification and programming framework.

2. Basic concepts

We follow closely the concepts of algebraic specifications, however, we are mainly interested in the aspect of functional enrichment. Therefore the notation of a signature $\Sigma = (S, F)$ consisting of a set S of sorts (names for carrier sets) and a set F of constants and function symbols decorated with sorts and functionalities (functional sorts defining the range and domain of the corresponding function) is used. In the following we assume a fixed set of sorts together with a fixed set of basic primitive functions. However, we allow not only simple sorts given by names, but also more refined sorts such as functional sorts and tuple sorts.

In specifications we assume a basic signature for which a Σ-algebra A is given. By A for every sort a carrier set is provided. We assume that the carrier sets contain the special element $\perp$ (called *bottom*) for formally representing the result of diverging computations and that every carrier set is partially ordered by an ordering $\sqsubseteq$ with $\perp$ as the least element and complete in the sense that there exist least upper bounds for directed sets. Furthermore we assume that all functions that are associated with the functions symbols of the signature in A are monotonic and continuous.

Predicate logic is an excellent tool for formal specification purposes. Let us consider a simple example to explain the basic idea of declarative specification. Consider for instance the formula

(*) $x = f(y)$

Clearly (*) is a simple formula of equational predicate logic. However, what does it actually specify? From the viewpoint of a computer scientist we would like to have more information about such a formula indicating how it is to be understood as a specification. This may be provided by sort or type information, such as

"x is of sort bool"

or

"y is of sort nat"

or

"f is of functionality (Nat $\rightarrow$ Bool)".

In addition, we may wish to give more information about the roles of x, y and f. For instance, the additional informations

"x is output"

and

"y is input"

or

"f is input"

give some "direction" for the specification and thus turns the formula of predicate logic into a specification of a computational task.

Typically in logic programming this information is not provided a priori in a logic program, but just provided in a query. Accordingly a parameter in a predicate in logic programming may change its role of carrying input or output. In declarative programming we are, however, interested in specifications and programs where the roles of identifiers as carrying input or output or serving as (implicitly) universally quantified are clearly stated.

When looking for a logical calculus for program development conciseness of notation becomes crucial. For these reasons we try to introduce in the following a number of notational concepts for shortening our formulas. If we write down a formula like

$$x + x = 2 * x$$

we may ask, whether this formula is universally or existentially valid for x. Like in the framework of algebraic specifications we may distinguish between the situation where x is part of the signature (and thus existentially quantified) or where x is not part of the signature, and therefore understood as universally quantified.

In formulas of many sorted predicate logic quantifiers are used for indicating three in principle independent logical properties of identifiers

- universal or existential validity,
- sort qualification ("typing"),
- abstraction (binding, hiding) of individual identifiers.

One of the fields of software development that is not fully mastered by nowadays methodology is programming-in-the-large. There typically a large number of data sets, elements and functions has to be considered and names for them have to be introduced ("declared"). The (notational) management of declarative specification and programming entities that introduce a large number of identifiers and logical or algorithmic properties for them therefore has to be considered crucial.

One of the problems of programming-in-the-large is the management of the rather long lists of identifiers. Therefore it seems just consequent to allow the abbreviation of such lists by identifiers. In addition, we consider lists (tuples) of elements. A list of (sorted) identifiers is defined by

$$X = (x_1 \in S_1 , ..., x_n \in S_n),$$

a list (a tuple) of elements is defined by

$$E = (e_1, ..., e_n).$$

By [X:=E] we define a substitution operator. It maps expressions and also formulas homomorphically onto expressions and predicate logic formulas resp. We write S1; S2 to compose substitutions. Note that in assertion logic procedural programs are treated by a pure substitution calculus.

Notational Conventions:
In the following the formulas of predicate logic are always interpreted within an assumed fixed sort-environment. A sort environment for the set of considered identifiers consists in a mapping of the identifiers to sorts. If in a binding construct no sort is explicitly given for an identifier, then for it the sort is taken that is associated with this identifier in the given sort-environment, or the sort is taken that is associated with the use of the identifier in the expression following the binding construct. If X is a (finite or infinite) set or tuple of identifiers we write

$$\forall X: t$$

(and also $\exists$ X: t) with the obvious meaning. For function application we often write f.x instead of the more common notation f(x). Note that we assume that the dot notation associates to the right. We write f.g.x for f(g(x)). $\square$

Based on the introduced notational concepts we next introduce the syntactic forms of declarative specifications.

3. Forms of Declarative Specifications

In the following we consider logical formulas that specify both a signature (i.e. a set of names for data elements and functions) and properties for the elements of that signature. To keep the approach simple we do not consider the introduction of new sorts such as found in algebraic specifications but rather concentrate on signatures which consist of a subset of the set X of identifiers for elements and functions decorated with their sorts

$$F \subseteq (X \times S)$$

where S is a given set of sorts and X is a set of identifiers. The pair (f, s) $\in$ F associates with the identifier f the particular sort s. In the following we assume that S contains all kinds of basic sorts as well as tuple sorts and functional sorts.

A declarative specification is a formula that characterizes a class of models. A model is given by a particular algebra with the resp. signature. However, in specifications and signatures we often want to distinguish between input and output. This means that we divide the signature declared by a formula into two parts: the input part and the output part. In addition we assume a number of presupposed primitives such as sorts and basic functions symbols for instance given by a classical algebraic specification.

For writing formulas we use classical predicate logic over the relational symbols included from the primitive signature such as equality. Apart from the classical forms of quantification we introduce two constructs that represent basic ideas of declarative specifications. Propositions such as

(1) "x is input of sort Nat"

or

(2) "y is output of sort Bool"

are represented by declarative logical formulas. In case (1) we write the declarative proposition

$\mathbb{I}\ x \in \text{Nat}$

and in case (2) we write the declarative proposition

$\mathbb{O}\ y \in \text{Bool}$

A declarative specification is a formula in predicate logic containing a number of declarative propositions. For indicating that declarative bindings are of a different nature than classical predicates we use a particular notation for them. We write declarative bindings similar to quantifications:

(*) $\qquad \mathbb{O}\ x \in M,$

(**) $\qquad \mathbb{I}\ x \in M.$

(*) and (**) are called *declarative propositions*. Formula (*) indicates that x of sort M is to be declared or to be provided as output, while (**) is a predicate which states the assumption that the object x of sort M is provided by the environment (as input). Note that by declarative propositions no classical logical information is provided, but additional "interface" information.

We avoid to write these formulas by propositions like "input(x)", since we are dealing here not with a logical statement about the *value* of identifier x, but talk about the *role* of identifier x. Given a logical statement t we write

$$\mathbb{O}\ x \in M: t,$$

$$\mathbb{I}\ x \in M: t.$$

Identifiers are essential in programming. The possibility to introduce names for certain mathematical elements and use them as abbreviations provides a lot of power for expressing algorithmic and logical concepts. However, in most approaches identifiers are treated only in a very static way. More flexibility when working with identifiers can lead to more adequate and more powerful notational concepts.

Example: Declarative specifications

(1) Declarative specification of division and modulo:

$$\mathbb{I}\ x, y \in \text{Nat}: \mathbb{O}\ d, r \in \text{Nat}: y > 0 \Rightarrow d*y = x+r \land 0 \le r < y$$

(2) Declarative specification of square root:

$$\mathbb{I}\ x \in \text{Nat}: \mathbb{O}\ y \in \text{Nat}: y^2 \le x < (y+1)^2$$ □

A declarative specification t is a formula in predicate logic which may contain quantifiers and declarative propositions. For an identifier x which is declared as input or output in a declarative statement t we may do bindings, too, as usual. In principle, we might even consider declarative specifications where the classification of identifiers as input or output is dynamic, i.e. depending on certain logical properties. However, for simplicity we assume and make sure in the following that the classification of identifiers as input or output is interpreted statically.

Declarative specifications are called *normal*, if they are in the following form (where t is a formula of predicate logic):

$$\mathbb{I}\ x_1 \in M_1, \ldots, x_n \in M_n: \mathbb{O}\ y_1 \in N_1, \ldots, y_m \in N_m: t$$

A formula in predicate logic represents a proposition for every assignment of values to the free identifiers in the formula. For a declarative specification this is similar. It always represents also a formula in predicate logic. However, in addition it specifies the role of certain free identifiers and thus a computational task.

We assume that the order in which the declarative specifications are written is not relevant. This is indicated by the following rules

$$\mathbb{O}\ Y: \mathbb{I}\ X: t = \mathbb{I}\ X: \mathbb{O}\ Y: t,$$

$$\mathbb{O}\ Y_A: \mathbb{O}\ Y_B: t = \mathbb{O}\ Y_A \cup Y_B: t,$$

$$\mathbb{I}\ Y_A: \mathbb{I}\ Y_B: t = \mathbb{I}\ Y_A \cup Y_B: t,$$

$$\mathbb{I}\ \emptyset: t = \mathbb{O}\ \emptyset: t = t.$$

The logical operations on formulas are extended to operations on declarative specifications by the following definitions:

$$\neg\mathbb{I}\ X: \mathbb{O}\ Y: t =_{df} \mathbb{I}\ X: \mathbb{O}\ Y: \neg t,$$

$$(\mathbb{I}\ X_A: \mathbb{O}\ Y_A: t_A) \lor (\mathbb{I}\ X_B: \mathbb{O}\ Y_B: t_B) =_{df} \mathbb{I}\ X_A \cup X_B: \mathbb{O}\ Y_A \cup Y_B: t_A \lor t_B,$$

$$(\mathbb{I}\ X_A: \mathbb{O}\ Y_A: t_A) \land (\mathbb{I}\ X_B: \mathbb{O}\ Y_B: t_B) =_{df} \mathbb{I}\ X_A \cup X_B: \mathbb{O}\ Y_A \cup Y_B: t_A \land t_B,$$

$$(\mathbb{I}\ X_A: \mathbb{O}\ Y_A: t_A) \Rightarrow (\mathbb{I}\ X_B: \mathbb{O}\ Y_B: t_B) =_{df} \mathbb{I}\ X_A \cup X_B: \mathbb{O}\ Y_A \cup Y_B: t_A \Rightarrow t_B,$$

$$\forall Z: \mathbb{I}\ X: \mathbb{O}\ Y: t =_{df} \mathbb{I}\ X \setminus Z: \mathbb{O}\ Y \setminus Z: \forall Z: t,$$

$$\exists Z: \mathbb{I}\ X: \mathbb{O}\ Y: t =_{df} \mathbb{I}\ X \setminus Z: \mathbb{O}\ Y \setminus Z: \exists Z: t.$$

By these rules every declarative specification can be brought into normal form.

For a given declarative specification t in normal form we may distinguish the following three sets of identifiers:

- FREE.t, the set of free identifiers,
- IN.t, the set of free identifiers designated as input,
- OUT.t, the set of free identifiers designated as output.

Note that we have

$$\text{IN.t} \subseteq \text{FREE.t}, \ \text{OUT.t} \subseteq \text{FREE.t}$$

Note furthermore that we do not assume that IN.t and OUT.t are disjoint.

In declarative specifications we assume that we consider formulas always in an environment with a declared signature. Let Σ be the actual signature. Given a formula

$$t$$

we write (following Dijkstra)

$$[t]$$

for

$$\forall\ (\text{IN.t} \cup \text{OUT.t}).$$

Substitutions can be used to change declarative propositions. The substitution rule for declarative propositions is given in the following (let x ynd y be distinct identifiers):

$$(\mathbb{O}\ x: t')[t/x] = \mathbb{O}\ \text{FREE.t}: t'[t/x],$$

$$(\mathbb{I}\ x: t')[t/x] = \mathbb{I}\ \text{FREE.t}: t'[t/x],$$

$$(\mathbb{O}\ x: t')[t/y] = \mathbb{O}\ x: t'[t/y],$$

$$(\mathbb{I}\ x: t')[t/y] = \mathbb{I}\ x: t'[t/y].$$

We understand declarative specifications as constructive ("algorithmic") or nonconstructive descriptions of tasks. A declarative specification is called a *complete interface* specification, if

$$\text{FREE.t} = \text{IN.t} \cup \text{OUT.t} \cup \Sigma.$$

A declarative specification t is called *fully satisfiable,* if

$$\forall\ \text{IN.t}: \exists\ \text{OUT.t}: t$$

A declarative specification t is called *partially satisfiable,* if

$$\exists\ \text{IN.t}: \exists\ \text{OUT.t}: t$$

A partially satisfiable specification can be understood to be meaningful only for certain input elements. The predicate

$$\exists\ \text{OUT.t}: t$$

is called the *implicit input restriction* of the declarative specification t. The implicit input restriction specifies for which input there does exist output.

Every declarative specification $\mathbb{I}$ X: $\mathbb{O}$ Y: t in normal form can be turned into a fully satisfiable interface specification by using instead the specification

$$(*) \qquad \mathbb{I}\ X: \mathbb{O}\ Y: (\exists\ Y: t) \Rightarrow t.$$

The specification (*) is called the *satisfaction closure* for $\mathbb{I}$ X: $\mathbb{O}$ Y: t. Trivially the satisfaction closure of any declarative specification t is fully satisfiable, since:

$$\forall\ \text{IN.t}: \exists\ \text{OUT.t}: (\exists\ \text{OUT.t}: t) \Rightarrow t$$

■

$$\forall\ \text{IN.t}: (\exists\ \text{OUT.t}: t) \Rightarrow (\exists\ \text{OUT.t}: t)$$

■ true

Note that the identifiers in OUT.t do not occur free in $\exists$ OUT.t: t. Moreover for a fully satisfiable declarative specification t the satisfaction closure yields t, again:

$$(\exists\ \text{OUT.t}: t) \Rightarrow t$$

■ $\qquad\qquad\qquad\qquad$ {($\exists$ OUT.t: t) ■ true}

true $\Rightarrow$ t

■ t

As a corollary we obtain that the satisfaction closure is an idempotent operation.

A declarative specification $\mathbb{I}$ X: $\mathbb{O}$ Y: t is called *underspecification,* if for some input the output is not uniquely determined.

As already mentioned we do allow for declarative specifications t that the set of input identifiers and the set of output identifiers are not disjoint, i.e. we allow that

$$\text{IN.t} \cap \text{OUT.t} \neq \emptyset.$$

This just means that some input appears also as output (also called *throughput*). In particular procedural programs and especially assignments can be understood as special forms of declarative specifications. There, however, we have to distinguish between the value of a variable in the input state and the value in the output state.

Assuming a complete partial order $\sqsubseteq$ for every carrier set sometimes the concept of *minimisation* of values of identifiers in formulas is useful. This corresponds to the possibility of writing inductive definitions: we write

$$\mu\ x \in S: t$$

for expressing that x is declared being output (i.e. $x \in \text{OUT}.\mu x \in S: t$) but from the set of possible output according to t only minimal output is taken (assume $y \notin$ FREE.t). Accordingly the formula above is assumed to be equivalent to the following formula:

$$\mathbb{O}\ x \in S: (t \wedge \forall\ y \in S: (t[y/x] \wedge y \sqsubseteq x \Rightarrow x \sqsubseteq y)).$$

The possibility of minimisation sometimes is useful in connection with specifications of least fixpoints. Note, however, that the satisfiability of this kind of formulas does only hold for specific formulas t.

There is a close relationship between declarative specifications and certain concepts of type theory. For instance product types or dependent types written by (let x and y be distinct)

$$y: (\Pi\ x: A)\ B$$

can be related to the predicate

$\mathbb{I}$ x: $\mathbb{O}$ y: x ∈ A ∧ y ∈ B.

However, in contrast to type theory we want to restrict our approach to a more syntactic treatment of sort and interface information and handle semantic properties by classical logical formulas and classical set theory.

4. Semantics of Declarative Specifications

As usual we consider a logical formula t with the set FREE.t of free identifiers semantically as a mapping V[t] associating with every environment for the free identifiers in t a truth value:

$$V[t]: (\text{FREE.t} \to D) \to \mathbb{B}$$

Here D is the considered universe of data. In this interpretation we ignore the existence of declarative propositions.

If we take declarative propositions into account semantically a declarative specification t can be understood as a predicate that specifies a mapping from valuations of its free identifiers FREE.t into a set of mappings from valuations for the input identifiers to valuations for the output identifiers. Accordingly its meaning is given by a mapping taking into account three sets of identifiers.

$$[\text{FREE.t}\backslash(\text{IN.t} \cup \text{OUT.t}) \to D] \to (([\text{IN.t}\to D] \to [\text{OUT.t}\to D]) \to \mathbb{B})$$

Note that IN.t ⊆ FREE.t, OUT.t ⊆ FREE.t is assumed. Note also the difference between a purely logical interpretation of a declarative specification (ignoring the information about the input and output character of the involved identifiers) and the interpretation as a declarative specification. For instance the declarative specification

$\mathbb{I}$ x ∈ Nat: $\mathbb{O}$ y ∈ Nat: x = y+1

is not fully satisfiable. For the valuation with

x = 0

there does not exist a valuation for y of the required property. Therefore there does not exist a mapping

$$\eta: ((\{x\}\to D) \to (\{y\} \to D))$$

such that for all $\sigma:\{x\}\to D$:

$$\sigma.x = (\eta.\sigma).y + 1$$

So in the spirit of declarative specifications the specification is equivalent to

$\mathbb{I}$ x ∈ Nat: $\mathbb{O}$ y ∈ Nat: false .

This is not true, of course, for the formula x = y+1.

Nevertheless we may use a satisfaction closure for only partially satisfiable specifications. Given the declarative specification $\mathbb{I}$ X: $\mathbb{O}$ Y: t we have defined its satisfaction closure by

$\mathbb{I}$ X: $\mathbb{O}$ Y: (∃ Y: t) ⇒ t.

In the case of our example above we get the following satisfaction closure:

$\mathbb{I}$ x∈Nat: $\mathbb{O}$ y∈Nat: (∃ y : x = y+1) ⇒ x = y+1

which is equivalent to:

$\mathbb{I}$ x ∈ Nat: $\mathbb{O}$ y ∈ Nat: x > 0 ⇒ x = y+1.

Note that by definition the satisfaction closure of a declarative specification is always fully satisfiable and free of input restrictions.

The declarative meaning of a fully satisfiable declarative specification t for every valuation for the free identifiers is given by a set of mappings

$$f: [\text{IN.t} \to D] \to [\text{OUT.t} \to D]$$

for every valuation of the free identifiers not in IN.t ∪ OUT.t. For every choice of values for the identifiers in FREE.t\(IN.t ∪ OUT.t) a declarative specification specifies mappings from valuations for the input identifiers onto valuations for the output identifiers.

Example: For the examples above we obtain that the formulas stand for predicates of the following form:

(1) $(\{x\} \to \mathbb{N}) \to ((\{d, r\} \to \mathbb{N}) \to \mathbb{B})$

(2) $(\{x\} \to \mathbb{N}) \to ((\{y\} \to \mathbb{N}) \to \mathbb{B})$

Note that the function space $(\emptyset \to M)$ is a one-element set and thus $(\emptyset \to M) \to \mathbb{B}$ is isomorphic to $\mathbb{B}$. □

Note that declarative specifications lead to a simpler mathematical structure concerning their semantics than predicate transformers (cf. [3]) where programs are formally objects of the set

$$\text{PROG} =_{df} [\text{STATE} \to [\text{STATE} \to \mathbb{B}]]$$

and for a given set ID of identifiers the set STATE is given by

$$\text{STATE} =_{df} [\text{ID} \to D]$$

and a predicate transformer is a function from

$$\text{PRED\_TRANS} =_{df} [\text{STATE}\to\mathbb{B}] \to [\text{STATE}\to \mathbb{B}]$$

Dijkstra uses the concept

$$wp: \text{PROG} \times [\text{STATE} \to \mathbb{B}] \to [\text{STATE} \to \mathbb{B}]$$

where

$$wp(p, q).\sigma = \forall\ \sigma' \in p.\sigma: \sigma \neq \Omega \wedge q.\sigma .$$

Here Ω denotes the state where all variables have "undefined" values.

5. Operations on Declarative Specifications

As general for specifying and programming large systems the available concepts for the composition of basic modules are decisive. For declarative specifications besides the already treated logical connectives we want to use the following operations

- renaming of input or output identifiers,
- hiding of input or output identifiers,
- sequential and parallel composition of declarative specifications as well as recursive declarations.

We may rename identifiers in declarative specifications just by using classical logical concepts. Given a set of disjoint identifiers $x_1, \dots x_n$, we rename these identifiers in the declarative specification t by writing

$$t[y_1 =: x_1, \dots, y_n =: x_n]$$

This can be understood just a shorthand (if we assume that no name-clashes between the y_i and the x_i occur) for

$$\exists\, x_1, \dots, x_n{:}\ x_1 = y_1 \wedge \dots \wedge x_n = y_n \wedge t.$$

Note that this form of renaming does not include renaming in declarative proposition.

Hiding can simply be expressed by existential quantification. Other operations on specifications are combining forms for declarative specifications. Examples are logical connectives, and

- sequential composition,
- parallel composition,
- fixpoint operators.

For declarative specifications A and B we define *sequential composition* by the following formula

$$A \circ B \equiv_{df} \exists\, (OUT.A \cap IN.B) \backslash (IN.A \cup OUT.B){:}\ A \wedge B$$

The *parallel composition* of A and B is defined by the conjunction

$$A \,||\, B \equiv_{df} \mu\, X{:}\ A \wedge B$$

where $X = (IN.A \cap OUT.B) \cup (IN.B \cap OUT.A)$.

Note that the sequential composition leads to a hiding („binding") of those identifiers that are in OUT.A and in IN.B, but not in IN.A or in OUT.B.

Given a declarative specification t and $x \in IN.t$ as well as $y \in OUT.t$ we denote the declarative specification obtained by the *feedback* of y to x by

$$t[x \leftarrow y].$$

This specification is defined to be equivalent to the following logical formula:

$$\exists\, x{:}\ t \wedge x = y.$$

Clearly sequential and parallel composition as defined here are very simple operations that may be expressed directly by the logical connectives.

Example: Composition of declarative specifications

The declarative specification (let x, y, z, a be distinct identifiers):

$$(\mathbb{I}\ x{:}\ \mathbb{O}\ y, z{:}\ t) \circ (\mathbb{I}\ z, a{:}\ \mathbb{O}\ r{:}\ t')$$

is by definition equivalent to

$$\mathbb{I}\ x, a{:}\ \mathbb{O}\ y, r{:}\ \exists\, z{:}\ t \wedge t'.$$

The declarative specification

$$(\mathbb{I}\ x, y{:}\ \mathbb{O}\ a, b{:}\ t) \,||\, (\mathbb{I}\ a, c{:}\ \mathbb{O}\ x, z{:}\ t').$$

is equivalent to

$$\mathbb{I}\ y, c{:}\ \mathbb{O}\ z, b{:}\ \mu\, x, a{:}\ t \wedge t' \qquad \square$$

Sequential and parallel composition can be used to express more programming language oriented forms of declarative specifications.

6. Naming Declarative Specifications

For reasons of simplicity of manipulation and notational convenience we also allow the "declaration" (introduction of names for) of declarative specifications. We assume a set Z of identifiers for declarative specifications. For simplicity we assume that Z and X, the set of identifiers for functions and elements considered so far, are disjoint. We write for a given declarative specification t

$$\textbf{spec}\ A \equiv t$$

to introduce A as a name for the declarative specification t. A can be used as an abbreviation for t in predicate specifications. We even allow that A is declared in a "recursive" manner.

Example: Recursive predicative specification

The following predicative specification A is recursive:

$$\begin{aligned}\textbf{spec}\ A \equiv\ & \mathbb{I}\ x, y{:}\ \mathbb{O}\ a, b{:}\ \exists\, z{:}\ A[z =: a] \wedge \\ & (C \Rightarrow a = g.z) \wedge (\neg C \Rightarrow a = h.z)\end{aligned} \qquad \square$$

A function τ is called a *predicate transformer*, if it takes a predicate as input and produces a predicate as output. In recursive declarations of predicative specifications of the form

spec $A \equiv \tau[A]$.

For A we define IN.A, OUT.A, FREE.A by the least sets (by set inclusion) that fulfil the following specifying equations:

$IN.A = IN.\tau[A]$,

$OUT.A = OUT.\tau[A]$,

$FREE.A = FREE.\tau[A]$.

τ is a predicate transformer. The theory of predicate transformers is studied for instance in [Dijkstra, Scholten 89]. If τ is monotonic w.r.t. implication "$\Rightarrow$", i.e. if

$[A \Rightarrow B] \Rightarrow [\tau[A] \Rightarrow \tau[B]]$,

then as well known τ has both a weakest and a strongest fixpoint.

A simple criterion for the monotonicity of $\tau[A]$ is given by the positivity of the occurrences of A: $\tau[A]$ is monotonic w.r.t. A if A occurs in the formula defining $\tau[A]$ only under an even number of negation signs.

For a specification language it is more difficult to decide which particular fixpoint to associate with recursive declarative specifications than for a programming language where operational concepts lead to clear constraints. For a specification language a liberal decision seems most appropriate. Therefore given a recursive declarative specification

spec $A = \tau[A]$

we associate with it any weakest specification that is a fixpoint of τ, i.e. A may stand for any specification with $A = \tau[A]$ such that for all B:

$[B = \tau[B]] \wedge [A \Rightarrow B] \Rightarrow [B \Rightarrow A]$.

Trivially if τ is monotonic then the specification associated with A is uniquely determined by the weakest fixpoint of τ.

Note that for specifying the strongest fixpoint of a function τ we may use a simple logical trick by writing:

spec $A' = \neg\tau[\neg A']$

spec $A = \neg A'$

We obtain from $\neg\tau[\neg A'] = A'$ and $A = \neg A'$

(1) $A = \neg\neg\tau[\neg A'] = \tau[A]$

and from $[B' = \neg\tau[\neg B']] \wedge [A' \Rightarrow B'] \Rightarrow [B' \Rightarrow A']$ with $B = \neg B'$:

$[B = \tau[B]] \wedge [B \Rightarrow A] \Rightarrow [A \Rightarrow B]$

This way we can also express that strongest fixpoints are associated with declarative specifications.

Examples: Named declarative specifications

(1) Let f be a function from $\mathbb{N}$ to $\mathbb{N}$:

spec $A \equiv \copyright\, f \in Nat \to Nat: \forall n \in Nat:$

$f.n = n \vee (n > 0 \wedge \exists f': A[f'=:f] \wedge f.n = f'(n\text{-}1))$.

By an analysis we obtain:

(*) $A \equiv \copyright\, f \in Nat \to Nat: \forall n: n \geq f.n$.

Trivially we can prove that A defined by (*) represents a fixpoint of the specifying equation:

$\forall n: f.n = n \vee$
$(n > 0 \wedge \exists f': \forall n: n \geq f'.n \wedge f.n = f'(n\text{-}1)) \equiv$

$\forall n: f.n = n \vee (n > 0 \wedge n\text{-}1 \geq f.n) \equiv$

$\forall n: n \geq f.n$.

To show that the formula (*) specifies the weakest fixpoint we have to prove that for all specifications B:

$[B \equiv \forall n: f.n = n \vee (n > 0 \wedge$
$\exists f': B[f'=:f] \wedge f.n = f(n\text{-}1))] \wedge$

$[A \Rightarrow B] \Rightarrow [B \Rightarrow A]$

It suffices to prove that from the fixpoint equation for B the predicate A follows: from

$B \equiv \forall n: f.n = n \vee$
$(n > 0 \wedge \exists f': B[f'=:f] \wedge f.n = f(n\text{-}1))$

we prove by induction on n:

$B \Rightarrow n \geq f.n$.

For $n = 0$ we obtain $f.0 = 0$. Now let $n > 0$ and the assumption $n' \geq f.n'$ hold for all $n' < n$. From B we obtain

$f.n = n \vee (n > 0 \wedge \exists f': B[f'=:f] \wedge f.n = f(n\text{-}1))$

The induction hypothesis gives

$f.n = 0 \vee (n > 0 \wedge f.n = f(n\text{-}1))$

which completes the proof.

(2) An example which is more difficult to express are choice functions on trees. A choice function gives as result for the argument t an arbitrary subtree of t or t itself. The following predicative specification A specifies such choice functions.

spec $A = \copyright\, f \in Tree \to Tree:$
$\forall t \in Tree: f.t = t \vee (\neg atomic.t \wedge$
$\exists f': A[f'=:f] \wedge (f.t = f'.car.t \vee f.t = f'.cdr.t))$

Interesting applications of recursive predicate specifications are obtained in the field of nondeterministic communicating systems. □

(3) Using the concept of recursive specifications we may simply write inductive definitions. For instance given a tree t the set of trees that are not subtrees can be defined by

spec Not_Subtree = $\mathbb{I}$ t ∈ Tree: $\mathbb{O}$ r ∈ Tree: t ≠ r ∧ Not_Subtree[right.r=:r] ∧ Not_Subtree[left.r=:r]

There, of course, a positive specification of the Subtree property would be more readable. □

Note that for nonmonotonic specifications extremal fixpoints may not exist even if fixpoints exist. Consider as an example the following specification:

spec A = ∃ x': ¬A[x'=:x].

Every specification of the form (for arbitrary n ∈ ℕ)

spec A_n = x < n

fulfils the specifying equation (take n for x'). However, the least upper bound of all A_n which is logically equivalent to true, does not fulfil the fixpoint equation. Such pathological examples can easily be avoided by restricting recursive specifications to monotonic predicates.

7. Declarative Specifications in the Large

Specifying and programming large systems is basically the discipline of composing of larger systems from given ones (or the decomposition of large systems into smaller ones) and of the management of many subsystems containing large numbers of components that are named in general by hopefully well chosen identifiers. Thus for programming in the large we need appropriate theories for modularisation, composition, and management of large numbers of subcomponents. A declarative specification is a generalisation and unification of units such as programs and their specifications that specify some input/output relations as well as specification units such as abstract data types, algebraic specifications etc.

When a program or a specification A is to be based hierarchically onto a given program or specification B this can easily be expressed by conjunction. We may write

A ∧ B.

Since we consider B as basis, we may assume

OUT.A ∩ IN.B = ∅

but even if the assumption is not true, we can do the composition of A and B.

8. Examples for Declarative Specifications

A function declaration is a simple basic example for a declarative program. It can be understood as a special case of a declarative specification: the declaration

fct f = (x ∈ M) M': t

is understood (if this declaration for f is not recursive) as equivalent to

$\mathbb{O}$ f ∈ M → M': ∀ x ∈ M: f.x = t.

In the case of recursion the function declaration is translated to

μ f ∈ M → M': ∀ x ∈ M: f.x = t.

More general, declarations of the form

let x ∈ M: x = t

are translated to

$\mathbb{O}$ x ∈ M: x = t

or in the case of recursion to

μ x ∈ M: x = t

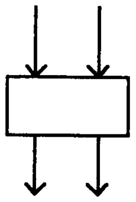
A declarative specification of the div and mod function is given by

spec Div_Mod ≡ $\mathbb{I}$ x, y ∈ Nat: $\mathbb{O}$ r, m ∈Nat:
y > 0 ⇒ x = y * r + m ∧ 0 ≤ m < y.

There are two ways of understanding a declarative specification t:

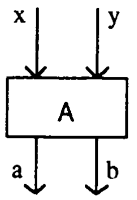
- a declarative specification can be understood to define a computational task t with input IN.t producing output OUT.t,
- a declarative specification can be understood as a definitional entity giving elements for the identifiers in OUT.t based on the identifiers in IN.t.

In the view of a declarative specification defining a computational task a declarative specification t can always be understood as a building block of the form

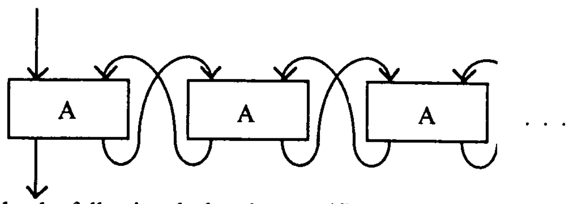


where the input lines correspond to input identifiers and the output lines correspond to output identifiers.

For instance let A be a specification corresponding to a building block with free variables x, y, a, b:



We may represent the infinite net



by the following declarative specification

spec NET = $\exists$ y, b: A $\wedge$ NET[b=:x, y=:a]

or using only minimal solutions by:

spec NET = (A || NET[b=:x, y=:a]).

Declarative specification can also be used to express programs or algorithms. A declarative program is called *algorithmic*, if it has the form

$\mathbb{I}$ x: $\mathbb{O}$ y: t

and t is *explicit in* y. A formula t is called *explicit in* y if it has the form

$(y = r_1 \vee \ldots \vee y = r_n)$

or (with y and z distinct)

$\exists$ z: t' $\wedge$ t"

and t' is explicit in y and t" is explicit in z. An interesting application for declarative specifications are switching circuits.

9. Concluding Remarks

In formalisms for the specification of mathematical elements and the description of algorithms the treatment of identifiers is crucial. One extreme is to work without any concept of free identifiers or binding such as combinatorial logic or functional programming in the style of Backus. Such an approach seems quite consequent and simple from a theoretical point of view. From a more practical point of view the flexible introduction of identifiers as abbreviations seems not only helpful, but the only way to master the quantitative and qualitative complexity of large programming tasks.

Assignment-oriented programming and assertion techniques are an example for such a logical treatment of computational tasks. However, using such concepts the treatment of identifiers is very specific. They generally are only used for denoting simple data values. Moreover, there one is forced to talk about their values in input states and their values in output states which makes reasoning technically more complicated. The other possibility is to talk about pairs of logical formulas such as pre- and post-conditions. This often forces one to use additional logical variables to be able to relate the input values of programming variables to output values.

Declarative specifications are understood as a tool for working with large sets of identifiers, including techniques of combining and renaming specifications. Here the binding mechanisms of λ-calculus and the quantifiers in predicate logic help to hide local identifiers. This way be a slight extension of the notation of logical formulas we obtain a flexible framework for the specification, design and verification of software modules.

References

[1] J. Backus: Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. CACM 28: 8, 1978, 613-641

[2] R. Bird: Lectures on Constructive Functional Programming. In: M. Broy (ed.): Constructive Methods in Computer Science. NATO ASI Series F: Computer and System Sciences Vol 55, Berlin-Heidelberg-New York: Springer 1989, 151-218

[3] E.W. Dijkstra: A Discipline of Programming. Englewood Cliffs, New Jersey: Prentice Hall 1976

[4] E.W. Dijkstra, C.S.Scholten: Predicate Calculus and Program Semantics. Springer 1989

[5] E.C.R. Hehner: Predicative Programming I + II. CACM 27:2 (1984) 134-151

[6] C.A.R. Hoare: Programs as Predicates. In: C.A.R.Hoare, J. C. Sheperdson (eds.): Mathematical Logic and Programming Languages. Prentice Hall 1984, 141-156.

[7] M.H. van Emden, R.A. Kowalski: The semantics of predicate logic as a programming language. J. ACM 23, 1976, 733 - 742.

[8] C.B. Jones: Systematic software development using VDM. Prentice Hall 1986