Implementation Specifications

Ch. Beierle, A. Voß

MEMO SEKI-85-08
IMPLEMENTATION SPECIFICATIONS

Christoph Beierle, Angelika Voß
Fachbereich Informatik, Universität Kaiserslautern
Postfach 3049, 6750 Kaiserslautern, West Germany

Abstract

Loose specifications of abstract data types (ADTs) have many non-isomorphic algebras as models. An implementation between two loose specifications should therefore consider many abstraction functions together with their source and target algebras. Just like specifications are stepwise refined to restrict their class of models, implementations should be stepwise refinable to restrict the class of abstraction functions. In this scenario specifications and implementations can be developed interwovenly. For example, we can consider implementations of sets by lists where the set simulating list operations are still left open. They may be refined later on so that an implementation of sets by arbitrary lists, by lists without double entries, or by sorted lists is obtained, differing e.g. in the efficiency of the set simulating operations.

We suggest to have implementation specifications analogously to loose ADT specifications: Implementations have signatures, models, axioms and sentences thus constituting an institution. Implementation specifications are the theories of this institution and refinements between implementation specifications are its theory morphisms.

1. Introduction

Fixed ADT specifications with only isomorphic models were studied before loose ADT specifications with non-isomorphic models, and several implementation concepts have been proposed, discussed, and revised for fixed specifications (e.g. [CTW 78], [Bec 82], [EK 78], [EKMP 82], [Ca 83]). By now there seems to be a basic consent that such an implementation concept should incorporate the following notions:

- an abstract specification to be implemented,
- a concrete specification implementing the abstract one,
- a signature morphism from the abstract to the (possibly extended) concrete specification allowing to translate abstract terms to concrete ones, and
- an abstraction function from the concrete to the abstract algebra allowing to translate the concrete value of a concrete term back to an abstract value.

Abstraction functions need not be totally defined, but must be surjective and homomorphic w.r.t. their domain of definition.

In contrast, so far only one implementation concept has been proposed for loose specifications that generalizes the fixed case, namely the concept proposed by Sannella and Wirsing in [SW 82]. Our own implementation concept generalizes that of Sannella and Wirsing giving room to a refinement process between implementations. Moreover, our approach abstracts from a particular ADT specification method by using the notion of an institution ([GB 83]) which provides abstract characterizations of signatures, models, sentences etc.

In Section 2 we outline the basic idea of our implementation concept. In Section 3 we briefly state the assumptions about the underlying loose specifications which are fundamental for our development. In Section 4 we introduce the institution of implementation specifications. In Section 5 we illustrate how implementations of sets by lists can be developed and refined stepwise hand in hand with the loose specifications. Section 6 contains a summary and a comparison.

2. Basic idea

As compared to fixed specifications, in the loose case we still have specifications, signatures, signature morphisms, etc, the essential difference lying in the number of models being considered. Therefore, an implementation for loose specifications should at least consist of
- an abstract specification,
- a concrete specification, and
- a signature morphism translating the abstract signature to the (possibly extended) concrete signature.

Since a concrete specification can always be extended before giving the implementation, we will choose the technically simpler approach and omit any extension of the concrete specification as part of the implementation.

Having translated an abstract term into a concrete one we are faced with the following questions:

1. In which concrete algebra shall the concrete term be evaluated, since there may be many non-isomorphic algebras?
2. To which abstract algebra shall we translate the value of the concrete term, since there also may be many non-isomorphic algebras?
3. Which abstraction function shall be used for the translation, since there may be different ones?

Having answered these questions we may further ask:

4. How can we specify the selected concrete algebras, abstract algebras, and abstraction functions in an implementation?

In [SW 82] Sannella and Wirsing require that for every concrete model there should be some abstract model and an abstraction function connecting them. If such a complete set of triples exists, the concrete specification is said to implement the abstract one, otherwise it does not. This is an implicit, non-constructive approach which gives no room for a notion of refinement between implementations since there is no way to characterize and restrict the set of triples - e.g. by constraints on the concrete or abstract models - any further.

Since the idea of loose specifications is to consider at first an arbitrary large set of models and to restrict this set stepwise by refining the specification, we think the adequate idea of implementations between loose specifications is to accept all meaningful combinations of an abstract model, a concrete model, and an abstraction function and to restrict them stepwise by refining the implementation.

To realize these ideas and answering (1) - (3) we introduce the notion of implementation models:

A simple implementation consisting of an abstract specification, a concrete one, and a signature translation between them denotes the set of all triples consisting of an abstract model, a concrete one, and an abstraction function from the concrete to the abstract model. Such a triple is called an implementation model. As in the fixed case, the abstraction function may be partially defined and it must be surjective and homomorphic.

Now we extend these simple implementations to a concept incorporating a notion of
refinement between implementations. Such a refinement should restrict the set of implementation models which can be done componentwise by
- restricting the set of abstract models,
- restricting the set of concrete models,
- restricting the set of abstraction functions.

In the framework of loose specifications the set of models - like the abstract and the concrete ones - is restricted by adding sentences to the respective specification. Thus we solve problem (4) w.r.t. the algebras by allowing sentences over the abstract and the concrete signature to be given in an implementation.

Since the abstraction functions operate on both concrete and abstract carriers we propose to view them as algebra operations from concrete to abstract sorts. These operations can be restricted as usually by adding sentences over both the concrete and the abstract signatures extended by the abstraction operation names. Thus we solve problem (4) completely by admitting arbitrary sentences over the abstract and the concrete signatures extended by the abstraction operation names. These sentences will be called implementation sentences.

Summarizing we propose an implementation specification to be
- a simple implementation
- together with a set of implementation sentences and
- denoting all implementation models of the simple implementation which satisfy the implementation sentences.

Analogously to specifications which consist of a signature in the simplest case, a simple implementation will also be called an implementation signature.

We already claimed that an implementation should be refinable by adding more implementation sentences to it and thus reducing the class of implementation models. This idea is extended analogously to loose ADT specifications by admitting a change of signature: There, a specification morphism is a signature morphism such that the translated sentences of the refined implementation hold in the refining specification.

Since an implementation contains two specifications, an implementation morphism should consist of two specification morphisms, an abstract one between the abstract specifications and a concrete one between the concrete specifications. With these two morphisms, the sentences of the refined implementation can be translated into sentences over the refining implementation by mapping the sorts and operations according to the two specification morphisms and by mapping the abstraction operation names to the corresponding abstraction operation names in the refining implementation.

Thus, a refinement between two implementations is given by an abstract and a concrete specification morphism such that the translated sentences of the refined implementation hold in the refining one.

7. The underlying institution of loose specifications

We only assume that the loose specifications have equational signatures with error constants, denote strict algebras, and are formally defined as the theories of an institution ([GB 83]).

Assumption: \( \text{SPEC-institution := } \langle \text{SIG}, \text{EAlg}, \text{ESen}, \sigma \rangle \)

is an institution where
4. The institution of implementation specifications

In order to develop our implementation concept in the framework of institutions we will have to make precise the notions of
- implementation signatures (Section 4.1) and
- implementation models (Section 4.2).

Having determined these notions we will establish a connection between loose specifications and implementations in Section 4.3 which will be helpful to formalize in Section 4.4 the remaining notions of
- implementation sentences and
- satisfaction of an implementation sentence by an implementation model.

Section 4.5 contains a summary of the new institution.

4.1 Implementation signatures

The signature \( \Sigma \) of a loose specification \( SP = \langle \Sigma, E \rangle \) may be viewed as a simple specification which has no sentences at all:
\[
\Sigma = \langle \Sigma, \emptyset \rangle.
\]
This suggested to define an implementation signature to be a simple implementation specification which has no sentences.

According to Section 2, such a simple implementation consists of an abstract specification \( SPA = \langle \Sigma a, E a \rangle \), a concrete specification \( SPC = \langle \Sigma c, E c \rangle \), and a signature morphism \( \sigma: \Sigma a \to \Sigma c \) translating the abstract to the concrete signature. Thus, an implementation signature, or shorter i-signature \( \mathcal{I} \), is a triple
\[
\mathcal{I} = \langle SPA, \sigma, SPC \rangle.
\]

We already suggested that a refinement between two implementations should consist of two specification morphisms between the abstract specifications and between the concrete specifications. Since an implementation comprises in particular an i-signature, we obtain the notion of refinement or morphism between i-signatures: An i-signature morphism
\[
\tau: \mathcal{I}_1 \to \mathcal{I}_2
\]
between two i-signatures \( \mathcal{I}_j = \langle SPA_j, \sigma_j, SPC_j \rangle \) for \( j \in \{1, 2\} \) is a pair
\[
\tau = \langle \sigma_a \sigma_c, \rho_a \rho_c \rangle
\]
consisting of an abstract specification morphism \( \sigma_a: SPA_1 \to SPA_2 \) and a concrete specification morphism \( \rho_c: SPC_1 \to SPC_2 \). The refinement requirement that the translated sentences of \( \mathcal{I}_1 \) must hold in \( \mathcal{I}_2 \) is trivially satisfied since \( \mathcal{I}_1 \) has no sentences at all.
However, another requirement should also be satisfied: Assume we have an i-signature from sets over arbitrary elements to extended lists over arbitrary elements, and another i-signature from sets over natural numbers to extended lists over natural numbers. Then it should not matter whether we first represent sets over arbitrary elements by lists over arbitrary elements and then refine to lists over natural numbers, or if we first refine the sets over arbitrary elements to sets over natural numbers and then represent them as lists over natural numbers.

In general that means that the specification morphisms $\sigma_1$ and $\sigma_2$ should be compatible with the signature morphisms $\alpha_1$ and $\alpha_2$. This in turn means that the diagram

$$
\begin{array}{ccc}
SPA_1 & \xrightarrow{\sigma_1} & SPC_1 \\
\downarrow \alpha_1 & & \downarrow \alpha_2 \\
SPA_2 & \xrightarrow{\sigma_2} & SPC_2
\end{array}
$$

should commute viewing $\alpha_1$ and $\alpha_2$ as signature morphisms.

These notions of i-signatures and i-signature morphisms constitute a category. In fact, it is the comma category induced by the functor $\text{Sig}$ forgetting specifications to their signatures.

**Definition 4.1** [ISIG, i-signature]

Given the forgetful functor $\text{Sig}: \text{SPEC} \to \text{Sig}$, the comma category

$$\text{ISIG} = (\text{Sig} \downarrow \text{Sig})$$

is the category of implementation signatures (i-signatures).

For an i-signature $\mathcal{I} = (\text{SPA}, \alpha, \text{SPC}) \in \text{ISIG}$, $\text{SPA}$ is called the abstract specification of $\mathcal{I}$, $\text{SPC}$ the concrete specification, and $\alpha$ the translation. For an i-signature morphism $\tau = (\text{pa}, \text{pc}) : \mathcal{I}_1 \to \mathcal{I}_2$, $\text{pa}$ is called the abstract specification morphism and $\text{pc}$ the concrete one.

Since the category $\text{SIG}$ is cocomplete and the functor $\text{Sig}$ preserves all colimits, $\text{ISIG}$ is cocomplete, too, by a general property of comma categories.

**Fact 4.2** [colimits]

$\text{ISIG}$ is cocomplete.

### 4.2 Implementation models

In Section 2 we already discussed the meaning of a simple implementation. Thus, for an i-signature $\mathcal{I}$ an $\mathcal{I}$-implementation model should consist of a concrete $\text{SPC}$-algebra $A_c$, an abstract $\text{SPA}$-algebra $A_a$, and a partial, homomorphic, surjective abstraction function $\alpha$ from $A_c$ to $A_a$. More precisely, for $A_a = (S_a, \text{Op}_{a})$ $\alpha$ is an $S_a$-indexed family of functions $\alpha_s$, each going into an abstract carrier $A_{as}$ and starting from the corresponding concrete carrier $A_{c(s)}$.

As in the fixed case, the concrete carriers and operations which are not needed for the translation can be forgotten along $\alpha$ so that we obtain $\text{EA}_{a}((\alpha)(A_c))$, i.e. $A_c$ viewed as a $E_a$-algebra, as the source of the abstraction function $\alpha$.

Now we could define $\alpha : \text{EA}_{a}((\alpha)(A_c)) \to A_a$ as a partial, surjective homomorphism. However, according to Section 2 we want to introduce abstraction operations as
ordinary operations which are interpreted by abstraction functions and which can be restricted by ordinary sentences. Since in the framework of the SPEC-institution the algebra operations must be totally defined, we will also require that the abstraction operations are totally defined. This is no limitation because the algebras are CPO:1 and there is an error constant for each sort denoting the minimum element. Thus \( a(x) \) is mapped to error whenever \( a(x) \) is meant to be undefined.

Doing so we must only suitably restrict the homomorphism requirement
\[
a(a(op)(x)) = op(a(x))
\]
which under these circumstances need to hold only if \( a(x) \) is non-error.

Calling a family of functions partially homomorphic if it is homomorphic except for the error elements, we can define an abstraction function
\[
a: EAlg(a)(Ac) \to Aa
\]
to be a surjective, partially homomorphic family of functions. Thus an \( \Sigma \)-implementation model, or just \( \Sigma \)-i-model \( MA \) is a tripe
\[
MA = \langle Ac, a, Aa \rangle.
\]

Note that in contrast to \( \Sigma \), where the first component is the abstract one and the third is the concrete one, we now have the abstract algebra in the third component and the concrete algebra in the first component. However, in both cases the first component contains the source and the third component the target of the function in the middle component.

Proceeding analogously to \( i \)-signature morphisms, we obtain a notion of \( i \)-model morphisms as a connection between two \( i \)-models. Given another \( i \)-i-model
\[
MB = \langle Bc, \delta, Ba \rangle
\]
an \( i \)-model morphism from \( MA \) to \( MB \) should consist of
- an \( SPa \)-homomorphism \( ha: Aa \to Ba \) and
- an \( SPc \)-homomorphism \( hc: Ac \to Bc \).

Analogously to \( i \)-signature morphisms the compatibility condition for \( i \)-model morphisms should express that it does not matter whether we first abstract \( Ac \)-elements with \( a \) to \( Aa \)-elements and then map them with \( hc \) to \( Bc \)-elements, or whether we first map the \( Ac \)-elements with \( hc \) to \( Bc \)-elements and then abstract them with \( \delta \) to \( Ba \). This condition may be expressed graphically by requiring that the square
\[
\begin{array}{ccc}
EAlg(a)(Ac) & \longrightarrow & Aa \\
| \downarrow \delta | \quad \quad \quad | \downarrow ha \\
EAlg(a)(hc) & \longrightarrow & Ba
\end{array}
\]
commutes. Note that we must forget \( hc \) along \( a \) because we also forget its source and target along \( a \).

To formalize this description we first have to solve a technical problem: In order to require that the square above commutes the morphisms occurring in there must belong to the same category. However, \( hc \) and \( ha \) are homomorphisms while \( a \) and \( \delta \) as abstraction functions are functions which are surjective but which are only partially homomorphic in general.

Since every homomorphism is partially homomorphic, but is not necessarily surjective the appropriate category for the square should have strict algebras as objects and partially homomorphic functions as homomorphisms.

**Definition 4.3** [\( \Sigma \)-p-homomorphism]
Let $A$, $B \in \text{EAlg}(\Sigma)$ with $\Sigma = \langle S,0,p \rangle \in \text{SIG}$. An $S$-sorted family of functions $h = \{h_s : A_s \rightarrow B_s \mid s \in S\}$ is a partially-homomorphic $\Sigma$-homomorphism (or just $\Sigma$-p-homomorphism) iff

$$
\forall s \in S : \forall x_1 \in A_{s_1} : \ldots : \forall x_n \in A_{s_n} : \begin{align*}
\text{hs}_1(x_1) \neq \text{error}_{-1B} & \land \ldots \land \text{hs}_n(x_n) \neq \text{error}_{-nB} \\
\Rightarrow h_{s}(\text{op}_A(x_1,\ldots,x_n)) = \text{op}_B(\text{hs}_1(x_1),\ldots,\text{hs}_n(x_n))
\end{align*}
$$

Fact 4.4 \text{ [p-homomorphisms are closed under composition]} \n
Let $\Sigma = \langle S,0,p \rangle \in \text{SIG}$ and $f : A \rightarrow B$, $g : B \rightarrow C$ be $\Sigma$-p-homomorphisms. Then their composition

$$g \circ f := \{g_s \circ f_s \mid s \in S\} : A \rightarrow C$$

is a $\Sigma$-p-homomorphism.

Definition 4.5 \text{ [PEAlg]} \n
The functor $\text{PEAlg} : \text{SIG} \rightarrow \text{CAT}$ maps a signature $\Sigma$ to the category of strict $\Sigma$-algebras with $\Sigma$-p-homomorphisms, and it maps a signature morphism $\sigma$ to the forgetful functor $\text{PEAlg}(\sigma)$ which is defined analogously to $\text{EAlg}(\sigma)$.

Definition 4.6 \text{ [Partial]} \n
For $\Sigma \in \text{SIG}$, $\text{Partial}_\Sigma : \text{EAlg}(\Sigma) \rightarrow \text{PEAlg}(\Sigma)$ is the inclusion functor.

Fact 4.7 \text{ [Partial is a natural transformation]} \n
$\text{Partial} : \text{EAlg} \Rightarrow \text{PEAlg}$ is a natural transformation.

With $\text{PEAlg}$ formalizing the property "partially homomorphic" we are now ready to define a preliminary model functor mapping an $\Sigma$-signature $\Sigma$ to the category of all triples $\text{TA} = \langle A_0, a, A_0 \rangle$ where $a$ is $\Sigma$-homomorphic but not necessarily surjective. Morphisms in that category are pairs of homomorphisms such that they commute with the abstraction functions. Similar to $\text{i}$-signatures, this situation can be expressed neatly as a comma category.

Definition 4.8 \text{ [Tripel(IE)]} \n
Let $\Sigma = \langle S_0, \sigma, S_0 \rangle$ be an $\Sigma$-signature with $\text{Sig}(S_0) = \Sigma$ and $\text{Sig}(S_0) = \Sigma$. The comma category $\text{Tripel}(\Sigma) := (\text{Partial}_{\Sigma} \circ \text{EAlg}(\Sigma) \downarrow \text{EAlg}(S_0) \downarrow \text{Partial}_{\Sigma} \circ \text{EAlg}(S_0))$ is called the category of $\Sigma$-tripels.

For an $\Sigma$-tripel $\text{TA} = \langle A_0, a, A_0 \rangle$, $A_0$ is called the abstract (or implemented) algebra, $A_0$ the concrete (or implementing) algebra, and $a$ is the abstraction function of $\text{TA}$.

Similar to ordinary signatures, every $\Sigma$-signature morphism induces a forgetful functor between the respective model categories in the reverse direction. It is defined componentwise.
Fact 4.9 [Tripel(τ)]

Let \( \tau = \langle p_a, p_c \rangle : \Sigma_1 + \Sigma_2 \in \text{SIG} \).

\( \text{Tripel}(\tau) : \text{Tripel}(\Sigma_2) \times \text{Tripel}(\Sigma_1) \)

defined on objects by

\( \text{Tripel}(\tau)(\langle A_c, a, A_a \rangle) := \langle E\text{Alg}(p_c)(A_c), P\text{Alg}(p_a)(a), E\text{Alg}(p_a)(A_a) \rangle \)

and on morphisms by

\( \text{Tripel}(\tau)(\langle h_c, h_a \rangle) := \langle E\text{Alg}(p_c)(h_c), P\text{Alg}(p_a)(h_a) \rangle \)

is a functor.

The observations above yield a preliminary model functor Tripel: \( \text{SIG} \rightarrow \text{CAT^{Top}} \). We still have to restrict this functor to consider only tripels with surjective abstraction functions.

Definition 4.10 [IMod(II:)]

For every \( \Sigma \in \text{SIG} \) the category of \( \Sigma \)-implementation models (or just \( \Sigma \)-im-models)

\( \text{IMod}(\Sigma) \)

is the full subcategory of Tripel(\Sigma) generated by all tripels with surjective abstraction function.

Fact 4.11 [IMod(τ)]

For every \( \tau : \Sigma_1, \Sigma_2 \in \text{SIG} \) the restriction and corestriction of Tripel(\tau) to IMod(\Sigma_2) and IMod(\Sigma_1) exist. It is denoted by

\( \text{IMod}(\tau) : \text{IMod}(\Sigma_2) \rightarrow \text{IMod}(\Sigma_1) \).

Definition 4.12 [IMod]

\( \text{IMod} : \text{SIG} \rightarrow \text{CAT^{Top}} \)

is the modelling functor for implementation signatures.

4.3 Relating implementation signatures to specifications

According to Section 2, implementation sentences over an i-signature \( \Sigma \) shall be expressed over the abstract signature \( \Sigma_a \), the concrete signature \( \Sigma_c \), and so-called abstraction operations to be interpreted as abstraction functions. In a first approach, implementation sentences will be all ordinary sentences over this vocabulary.

With this decision we can define the set of \( \Sigma \)-implementation sentences to be the set of all ordinary \( \psi(\Sigma) \)-sentences, where \( \psi(\Sigma) \) is a suitable equational signature combining \( \Sigma_a \), \( \Sigma_c \), and the abstraction operations.

We must define \( \psi(\Sigma) \) such that the sentences which are to restrict the abstract algebras do not affect the concrete ones and vice versa. Therefore, taking the set theoretic union of signatures is not suitable, since the abstract and the concrete signatures need not be disjoint. Thus we take the disjoint union (or coproduct). Moreover, for reasons of convenience we will use standard names for the abstraction operations:

Definition 4.13 [abs-operations]

For \( \Sigma = \langle \Sigma_a, \Sigma_c, \Sigma_c \rangle \in \text{SIG} \) and \( \tau = \langle p_a, p_c \rangle : \Sigma + \Sigma' \in \text{SIG} / \) we define:

\( \text{abs-operations}(\Sigma) := \{ \text{abs-s}_{\Sigma}: \sigma(s) + s \mid s \in \Sigma_a \} \)

\( \text{abs-operations}(\tau) := \{ (\text{abs-s}_{\Sigma}, \text{abs-p}_a(s))_{\Sigma'} \mid s \in \Sigma_a \} \).
Fact 4.14 \([\psi]\)

\[ \psi : ISIG \times SIG \]

defined on objects by
\[ \psi(\Pi) := Ia \times Ec \uplus \text{abs-operations}(\Pi) \]

and on morphisms by
\[ \psi(\tau) := pa \times pc \uplus \text{abs-operations}(\tau) \]

is a colimit preserving functor.

Having defined an IE-implementation sentence to be an ordinary \(\psi(\Pi)\)-sentence \(p\) we must determine whether an IE-i-model \(MA = <Ac,\alpha,AA>\) satisfies \(p\). Since the abstract symbols in \(\psi(\Pi)\) shall be interpreted by the abstract algebra \(AA\), the concrete symbols by the concrete algebra \(Ac\), and the abstraction operations by the abstraction function \(\alpha\), we can take the disjoint union of \(AA\), \(Ac\), and \(\alpha\) to obtain a \(\psi(\Pi)\)-algebra interpreting \(p\).

Definition 4.15 \([\text{join } I\Pi(\Pi MA)]\)

For an i-signature \(I\Pi = <SPa,\alpha,SPc>\) and an IE-i-model \(MA = <Ac,\alpha,AA>\)

\[ \text{join }_I\Pi(\Pi MA) := AA \uplus Ac \uplus \alpha \]

is the \(\psi(\Pi)\)-algebra \(A\) defined by
- for \(s \in \text{Sig}(SPa)\):
  \[ A_s := As \]
- for \(s \in \text{Sig}(SPc)\):
  \[ A_s := Ac_s \]
- for \(op \in \text{Sig}(SPa)\):
  \[ opA := opAs \]
- for \(op \in \text{Sig}(SPc)\):
  \[ opA := opAc \]
- for \(abs-s \in \text{abs-operations}(\Pi)\):
  \[ abs-s := _A s \]

The join operator can be extended to a functor from IE-i-models to \(\psi(\Pi)\)-algebras.

Fact 4.16 \([\text{join } I\Pi]\)

Defining \(\text{join }_I\Pi\) on IE-i-model morphisms \(\rho = <hc,ha>\) by

\[ \text{join }_I\Pi(\rho) := \{ ha_s \mid s \in \text{Sig}(SPa) \} \uplus \{ hc_s \mid s \in \text{Sig}(SPc) \} \]

yields a functor

\[ \text{join }_I\Pi : \text{IMod}(\Pi) \to \text{EAlg}(\psi(\Pi)) \]

Generalizing over all i-signatures we obtain a natural transformation from the implementation model functor to the model functor of the SPEC-institution composed with the signature translation.

Fact 4.17

\[ \text{join} : \text{IMod} \Rightarrow \text{EAlg} \circ \psi \]

is a natural transformation.

Now the question whether an IE-i-model \(MA\) satisfies an implementation sentence \(p\) has been reduced to the question whether \(\text{join }_I\Pi(\Pi MA)\) satisfies \(p\) in the framework of the SPEC-institution.

4.4 Implementation sentences and their satisfaction

According to the preceding section we define the set of IE-implementation sentences or just IE-i-sentences to be the set of all ordinary \(\psi(\Pi)\)-sentences. Such an IE-i-sentence \(p\) is satisfied by an IE-i-model \(MA\) exactly if \(MA\) viewed as the \(\psi(\Pi)\)-algebra \(\text{join }_I\Pi(\Pi MA)\) satisfies \(p\).
Definition 4.18 \([\text{ISen}]\)

The implementation sentence functor is given by

\[ \text{ISen} := \text{Sen} \circ \phi : \text{SIG} \rightarrow \text{SET}. \]

Definition 4.19 \([\mathbb{1}]\)

Let \( E \in \text{SIG}, \ MA \in \text{IMod}(E) \) and \( p \in \text{ISen}(E) \). \( MA \) satisfies \( p \), written \( \models_{\text{ISIG}}^E \ MA \models p \)

iff \( \text{join}_{E}(MA) \models_{\psi(E)} p \).

Fact 4.20 \([\text{saturation condition}]\)

\[ \forall \tau : E_1 \rightarrow E_2 \in \text{SIG}, \ \forall MA \in \text{IMod}(E_2), \ \forall p \in \text{ISen}(E_2), \ \forall MA \models_{E_2} \text{ISen}(\tau)(p) \iff \text{IMod}(\tau)(MA) \models_{E_1} p. \]

4.5 The institution

Since the satisfaction condition holds the notions defined above constitute an institution.

Definition 4.21 \([\text{IMP-institution}]\)

\[ \text{IMP-institution} := \langle \text{ISIG}, \text{ISen}, \text{IMod}, \mathbb{1} \rangle \]

is the institution of implementation specifications.

Like specifications are defined as the theories of the SPEC-institution, implementation specifications will be defined as the theories of this new institution.

Definition 4.22 \([\text{IMP}]\)

\text{IMP} is the category of theories of the IMP-institution and it is called the category of implementation specifications.

Thus an implementation specification or just i-specification ISP is a pair

\[ \text{ISP} = \langle E, \text{ISig} \rangle \]

consisting of an i-signature \( E \) and a set of \( E \)-i-sentences \( \text{ISig} \), and an i-specification morphism is an i-signature morphism respecting the i-sentences.

Since \( \text{SIG} \) is cocomplete, general institution properties tell us that \text{IMP} is cocomplete as well.

Fact 4.23 \([\text{colimits}]\)

\text{IMP} is cocomplete.

5. Examples: developing implementations of sets by lists

In our examples we will assume that the error constants are implicitly declared. As sentences we will use first order formulas where the bound variables are not interpreted as bottom elements. Besides we need some constraint mechanism to exclude
unreachable elements (e.g. initial [HKR 80], data [BG 80], hierarchy [SW 82], or algorithmic constraints [BV 85]).

We will show how several well known implementations of sets by lists can be developed stepwise and hand in hand with the implementing specification.

On the abstract side we have the specification SET of sets with the empty set as constant, and operations to insert an element, to determine or remove the minimum element in a set, and to test for the empty set or for the membership of an element. Besides standard sets, there may be bags or unreachable elements of sort set. The set elements are described in the specification LIN-ORD which introduces a sort elem with an equality operation and an arbitrary reflexive linear ordering. The subspecification BOOL of LIN-ORD specifies the booleans.

On the concrete side the specification LIST extends LIN-ORD to standard lists with the constant nil, the operations cons, car, and cdr, and a test nil? for the empty list. All lists must be generated from the elements by nil and cons. LIST is extended to LIST-S by introducing names for the set simulating operations, but without restricting these operations in order to obtain a variety of different models.

```
spec BOOL =
  sorts bool
  ops  true, false: -> bool
       not: bool + bool
       and, or: bool bool + bool
  sentences ... < specifying the booleans >

spec LIN-ORD = BOOL u
  sorts elem
  ops  eq, le: elem elem + bool
  sentences ... < specifying eq as equality and le as an arbitrary reflexive linear ordering >

spec SET = LIN-ORD u
  sorts set
  ops  empty: + set
       insert: elem set + set
       min: set + elem
       remove-min: set + set
       empty?: set + bool
       in?: elem set + bool
  sentences ... < specifying the set operations with their usual meaning, but not necessarily excluding non-standard sets >

spec LIST = LIN-ORD u
  sorts list
  ops  nil: + list
       cons: elem list + list
       car: list + elem
       cdr: list + list
       nil?: list + bool
  sentences ... < specifying standard lists over elem generated by nil and cons >

spec LIST-S = LIST u
  ops  l-insert: elem list + list
       l-min: list + list
       l-remove-min: list + list
       l-in?: elem list + bool
```

Figure 5.1 The ADT specifications in the implementation of sets by lists
Presentations of the specifications mentioned so far are given in Figure 5.1. The sentences parts are not elaborated since the necessary first order formulas are standard and since we did not want to go into the details of the constraint mechanism to be used, because our implementation concept abstracts from these details completely.

We can give a first simple i-specification I:SET/LIST-S from SET to LIST:

\[
\text{i-spec } I: \text{SET}/\text{LIST-S} = \\
i\text{sig } \sigma_{S/LS}: \text{SET} \to \text{LIST-S}
\]

with the signature morphism

\[
\sigma_{S/LS}: \text{Sig(SET)} \to \text{Sig(LIST-S)}
\]

\[
\begin{align*}
\text{set} & \to \text{list} \\
\text{empty} & \to \text{nil} \\
\text{empty?} & \to \text{nil?} \\
\text{insert} & \to \text{l-insert} \\
\text{in?} & \to \text{l-in?} \\
\text{min} & \to \text{l-min} \\
\text{remove-min} & \to \text{l-remove-min} \\
x & \to x & \text{for } x \in \text{Sig(LIN-ORD)}
\end{align*}
\]

It merely defines the signature morphism \(\sigma_{S/LS}\) translating sort set to list and translating the set operations to their simulating list operations without renaming the signatures of the common subspecifications LIN-ORD and BOOL. Since I:SET/LIST-S contains no i-sentences, its i-models comprise all possible implementations of sets by lists.

I:SET/LIST-S can be refined in various ways by adding i-sentences restricting the abstraction operations of sort set, such that e.g.

\[
\text{i-spec } I\text{A:SET}/\text{LIST-S} = I: \text{SET}/\text{LIST-S u isentences}
\]

\[
(\forall x: \text{list . } \forall e: \text{elem.} \\
\text{abs-set}(\text{cons}(e,x)) = \text{insert}(\text{abs-elem}(e),\text{abs-set}(x)))
\]

\[
\text{i-spec } I\text{S:SET}/\text{LIST-S} = I: \text{SET}/\text{LIST-S u isentences}
\]

\[
(\forall e, e1, e2: \text{elem . } \forall x: \text{list .} \\
\text{abs-set}(\text{cons}(e,\text{nil})) = \text{insert}(\text{abs-elem}(e),\text{empty}) \land \\
\text{le}(e1,e2) = \text{true } \& \text{eq}(e1,e2) = \text{false } \Rightarrow \\
\text{abs-set}(\text{cons}(e1,\text{cons}(e2,x))) = \\
\text{insert}(\text{abs-elem}(e1),\text{abs-set}(\text{cons}(e2,x))) \land \\
\text{le}(e2,e1) = \text{true } \& \text{eq}(e1,e2) = \text{false } \Rightarrow \\
\text{abs-set}(\text{cons}(e1,\text{cons}(e2,x))) = \text{error-set})
\]

\[
\text{i-spec } I\text{U:SET}/\text{LIST-S} = I: \text{SET}/\text{LIST-S u isentences}
\]

\[
(\forall e, e1, e2: \text{elem . } \forall x: \text{list .} \\
\text{abs-set}(\text{cons}(e,\text{nil})) = \text{insert}(\text{abs-elem}(e),\text{empty}) \land \\
(\text{in?}(e,\text{abs-set}(x)) = \text{true } \Rightarrow \\
\text{abs-set}(\text{cons}(e,x)) = \text{error-set})
\]

\[
\text{i-spec } I\text{SU:SET}/\text{LIST-S} = I\text{U:SET}/\text{LIST-S u IS:SET}/\text{LIST-S}
\]

Figure 5.2  Some i-specifications implementing sets by lists
- all lists represent sets (IA:SET/LIST-S),
- only lists with unique entries may represent sets (IU:SET/LIST-S),
- only sorted lists may represent sets (IS:SET/LIST-S), or
- only sorted lists with unique entries may represent sets (ISU:SET/LIST-S).

The last i-specification refines not only I:SET/LIST-S, but also IU:SET/LIST-S and IS:SET/LIST-S. The i-specifications are given in Figure 5.2 where we use abs-s: \( \sigma_S/LS(s) + s \) as the abstraction operation name of sort \( s \).

By restricting the abstraction operations these alternative i-specifications constrain their i-models not only w.r.t. the abstraction function of sort set, but also w.r.t. the set simulating list operations. Correspondingly, we can specify four refinements of the concrete LIST-S specification by adding sentences fixing the set simulating operations, such that they generate (and operate upon) exactly
- all lists (LIST-SA),
- all lists with unique entries (LIST-SU),
- all sorted lists (LIST-SS), or
- all sorted lists with unique entries (LIST-SSU).

To give an example we elaborate the specification LIST-SS:

\[
\text{spec LIST-SS = LIST-S u sentences}
\]
\[
\begin{align*}
1\text{-min}(\text{nil}) &= \text{error-elem} \\
1\text{-remove-min}(\text{nil}) &= \text{nil} \\
(\forall e: \text{elem} . \; 1\text{-insert}(e,\text{nil}) = \text{cons}(e,\text{nil})) \\
(\forall x: \text{list}, \; \forall e: \text{elem} . \; \\
\begin{align*}
1\text{-min}(\text{cons}(e,x)) &= e & \& \\
1\text{-remove-min}(\text{cons}(e,\text{cons}(e,x))) &= 1\text{-remove-min}(\text{cons}(e,x)) & ) \\
\end{align*}
\end{align*}
\]
\[
(\forall x: \text{list}, \; \forall e1, e2: \text{elem} . \; \\
\begin{align*}
e(e1,e2) &= \text{true} \Rightarrow \\
1\text{-insert}(e1,\text{cons}(e2,x)) &= \text{cons}(e1,\text{cons}(e2,x)) \& \\
1\text{-in?}(e1,\text{cons}(e2,x)) &= \text{eq}(e1,e2)) \& \\
(eq(e1,e2) &= \text{false} \Rightarrow \\
1\text{-insert}(e2,\text{cons}(e1,x)) &= \text{cons}(e1,1\text{-insert}(e2,x)) \& \\
1\text{-in?}(e2,\text{cons}(e1,x)) &= 1\text{-in?}(e2,x) \& \\
1\text{-remove-min}(\text{cons}(e1,\text{cons}(e2,x))) &= \text{cons}(e2,x) \) \\
\end{align*}
\]

Now we can in turn refine each of the i-specifications IX:SET/LIST-S for \( X \in \{ A, U, S, SU \} \) by replacing the concrete specification LIST-S by its refinement LIST-XS and calling the resulting i-specifications IX:SET/LIST-XS. Since the abstraction operation of sort elem is not restricted, IX:SET/LIST-XS i-models have many non-isomorphic LIN-ORD implementations, e.g. the characters represented by the natural numbers or by the integers. However, for each LIN-ORD implementation there are only isomorphic IX:SET/LIST-XS i-models as extensions since the set simulating operations are fixed by now. Thus
- IX:SET/LIST-SA specifies the implementation of sets by all lists,
- IX:SET/LIST-SU specifies the implementation of sets by all lists with unique entries,
- IX:SET/LIST-SS specifies the implementation of sets by all sorted lists, and
- IX:SET/LIST-SSU specifies the implementation of sets by all sorted lists with unique entries,

which are well known sets-by-lists implementations, differing in the time efficiency of the set simulating operations and in the amount of storage needed by the lists. As an example we give the i-specification IX:SET/LIST-SS:
The refinement relations between the specifications and i-specifications described above are depicted in Figures 5.3 and 5.4:

6. Conclusions

We proposed an implementation concept for loose abstract data type specifications. It introduces the notions of implementation signatures, - models, and - specifications and it formalizes the transition from a more abstract to a more concrete specification including a change of the underlying data structures. By providing the notion of refinement between implementations it supports to develop specifications and implementations hand in hand. Moreover, using institutions our concept abstracts from the underlying ADT specification method and is applicable to different specification techniques.

Other implementation concepts for loose specifications lack the notion of refinement. An implementation in the approach for Clear-like specifications proposed in [SW 82] is - in our terminology - an i-signature with the semantic condition that for every abstract algebra there is a concrete one with an abstraction function in between. Concepts like those of [GM 82] and [Sch 82] are based on behavioural abstraction and have been proposed for modules. The implementation concept for the kernel language
ASL of [SW 83] merely requires that the abstract specification is included in the concrete one. This simple notion is based on the fact that, as a semantical language, ASL has very powerful specification building operations which however may not be present in a language for ADT specifications.

In [BV 85] the implementation concept proposed here is elaborated for a particular institution of ADT specifications. An effective procedure is given to convert an implementation specification to a normal form which essentially consists of an ordinary ADT specification. Referring to the normal forms associative vertical composition operations for implementations are defined compatibly on the syntactical and semantical levels. Horizontal composition and instantiation of parameterized implementations are compatible with vertical composition allowing to combine implementation specifications interchangeably in different directions with the same result.

References


