Data Conversions in Abstract Data Types

Nico Verwer

RUU-CS-88-30
September 1988

Rijksuniversiteit Utrecht

Vakgroep informatica

Padualaan 14 3584 CH Utrecht
Corr. adres: Postbus 80.089, 3508 TB Utrecht
Telefoon 030-531454
The Netherlands

Data Conversions in Abstract Data Types

Nico Verwer

Technical Report RUU-CS-88-30
September 1988

Department of Computer Science
University of Utrecht
P.O.Box 80.089, 3508 TB Utrecht
The Netherlands

Data Conversions in Abstract Data Types

Nico Verwer

Abstract

Loose algebraic specifications can have many non-isomorphic models, or implementations. If data is transported between different implementations of the same specification, its representation has to be changed. This task is performed by a data conversion program.

We review the theory of algebraic specifications, and present a notion of programs. A correctness condition for data conversions is proposed, and some of its properties are investigated. The notion of data conversion is generalized to that of abstract data conversion, in order to be able to translate between arbitrary models of a specification. Finally we point at some aspects of constructing data conversions for a particular specification.

Contents

1 Introduction 2

2 Specifications, models and programs 3

3 Data conversions between different models 9

4 Abstract data conversions 19

5 Implementations and data conversions 22

6 Further research 26

1 Introduction

An important feature of algebraic specification techniques is the possibility of *loose* specifications. Loose specifications leave out all details concerning the implementation, and they allow a large class of models (algebras)to be correct with respect to the semantics of the specification language. The model class is called the abstract data type corresponding to the specification.

All algebras in the model class of a specification are equivalent in an abstract sense, but they certainly do not have to be equal (by equal we mean 'equal up to isomorphism'). Differences between models arise because of different implementation decisions, such as different representations of a data type, or different algorithms to perform some computation. These differences manifest themselves as different equalities that hold in models, e.g. two terms t and t' may be represented as one object in one model, and as two different objects in another. Also there may be terms that do have a representation in one model but not in another (i.e. they are undefined in that model).

The task of the implementor of a specification SP_0 is to refine this via a number of steps $\mathrm{SP}_1, \ldots, \mathrm{SP}_n$, finally leading to an implementation SP_n, which is a specification with sorts and functions that are of a sufficiently low level to be executable on some existing machine. Another programmer however, may implement SP_0 differently, arriving at an implementation SP'_m, along a sequence $\mathrm{SP}'_1, \ldots, \mathrm{SP}'_m$ of intermediate implementations. In this case the *meaning* of data in SP_n and SP'_m is the same, but its representation may be very different.

As an example, consider a specification DB_0 of a database system, and two different implementations DB_n and DB'_m. For some reason, it may be desirable to transport data represented according to one model to the other. An obvious way of changing the representation is to translate data in DB_n into a DB_0 term representation, which might be a list of database-transactions. Then the representation in DB'_m can be built from this list, by feeding it as an input to the DB'_m implementation. This process could be made less time-consuming if a program were written to convert the data 'directly', without an intermediate SP_0-representation.

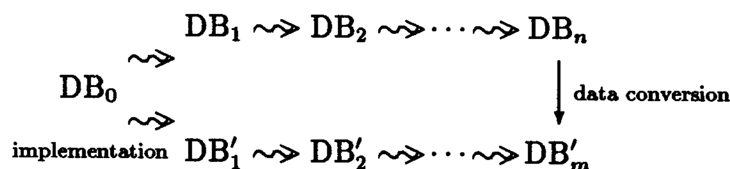

Figure 1: Two implementations of DB_0

Writing a representation-changing program by hand may be quite cumbersome, and does not seem to be worthwhile, unless it is used often. However, if the implementation is done in a framework of formal specifications, such a *data conversion* program can be constructed from the information recorded in the implementation steps $\mathrm{SP}_1, \ldots, \mathrm{SP}_n$ and $\mathrm{SP}'_1, \ldots, \mathrm{SP}'_m$. Because these steps are documented in a formalized way, the construction of the data conversion becomes a rather mechanical process, amenable to automation.

It is our aim to provide conditions for the automatic construction of data conversion programs. Under these conditions it is possible to derive a data conversion, using the information recorded during the implementation process.

In section 2 we briefly review the theory of algebraic specifications, and introduce a way of constructing programs in this framework. The core of the article is section 3, where we develop a notion of correctness for data conversions between models of a particular specification, and look at some of their properties. This is generalized in a rather straightforward way to abstract data conversions in section 4. In section 5 we briefly look at a theory of implementations, and point at some aspects of finding abstract data conversions between constructor-implementations.

A shorter version of this paper has appeared in [7]. All important additions are marked by an asterisk (*) in the margin. *

2 Specifications, models and programs

It is assumed that the reader is acquainted with the basic notions used in the theory of algebraic specifications and implementations. Therefore we only briefly review some concepts and notations, to fix the framework in which we shall develop the theory of data conversions. Those who are new to algebraic specifications should consult [2], [4] or any other introductory text.

All the expressions (terms) we use are presumed to be well formed, i.e. all subexpressions are of the right type. For any set S, by S^* we denote the cartesian product $S \times \cdots \times S$, where the exact number of components is determined by the context in which S^* appears.

A signature Σ is a pair $\langle \Sigma_{\mathbf{sorts}}, \Sigma_{\mathbf{opns}} \rangle$, where the first element is a set of sort names, and the second element is a $(\Sigma^*_{\mathbf{sorts}}, \Sigma_{\mathbf{sorts}})$-sorted set of operation symbols. The category **Sign** contains all signatures as objects, and the signature morphisms $\sigma = \langle \sigma_{\mathbf{sorts}}, \sigma_{\mathbf{opns}} \rangle$ as morphisms.

We denote the carrier set for sort $\mathbf{s}$ of an algebra A by $\mathbf{s}^{\mathrm{A}}$, and the function belonging to an operation symbol f by f^{A}. The union of all carrier sets is denoted by $|A|$.

The ground term algebra over a signature Σ, denoted by T_Σ, is constructed in the usual way. For $t \in \mathrm{T}_\Sigma, t^{\mathrm{A}}$ is the interpretation of t in A. The set of terms with variables $(x_1, \ldots, x_n)$, is the term-algebra $\mathrm{T}_\Sigma(x_1, \ldots, x_n)$. If $t \in \mathrm{T}_\Sigma(x_1, \ldots, x_n)$, we can substitute the terms u_i for the variables x_i; the resulting term is written as

$t[(u_1,\ldots,u_n)/(x_1,\ldots,x_n)]$.

As we consider partially defined algebras, we need the definedness predicate $\mathbf{D}$ over terms. For every term $t \in T_\Sigma$, we add the sentence $\mathbf{D}(t)$ to our logical system. For a particular term t, the fact that $\mathbf{D}(t)$ holds in the algebra A is expressed as A by $A \models \mathbf{D}(t)$ or, equivalently, $\mathbf{D}(t^A)$. If $\mathbf{D}(t)$ can be proved from the specification, we write $SP \vdash \mathbf{D}(t)$.

Like [1], we define equality between partially defined terms as 'strong equality': If t and t' are terms of the same sort

$$A \models t = t' \Leftrightarrow (A \not\models \mathbf{D}(t) \wedge A \not\models \mathbf{D}(t')) \vee (A \models \mathbf{D}(t) \wedge A \models \mathbf{D}(t') \wedge t^A = t'^A)$$

We only consider functions f^A that are strict; if $f^A(x_1,\ldots,x_n)$is defined, all of its parameters $x_1,\ldots,x_n$ must be defined. We shall exclude higher-order functions and non-determinism.

A specification is a sentence in a particular language. We are not interested in its specific syntactic form, but we assume that it has a semantics, given by the maps

$$Sig[\![SP]\!] \in \mathsf{Sign}$$

the signature, and

$$Mod[\![SP]\!] \subseteq \mathsf{PAlg}(Sig[\![SP]\!])$$

the model class ($\mathsf{PAlg}(\Sigma)$ is the category of all partial Σ-algebras).

Because a loose specification alows of a large model class, we cannot restrict our attention to initial or final algebra semantics. Therefore we only require that the specification language has a sound (but not necessarily complete) proof system, i.e.

$$SP \vdash \epsilon \Rightarrow Mod[\![SP]\!] \models \epsilon$$

Here ϵ is some sentence in our logical system, $SP \vdash \epsilon$ means that it is provable, and $Mod[\![SP]\!] \models \epsilon$ means that it holds in every model in $Mod[\![SP]\!]$.

Let A and B be Σ-algebras. A family $\phi = \left\{\phi_\mathbf{s} : \mathbf{s}^A \rightarrow \mathbf{s}^B\right\}$ of mappings from A to B is a homomorphism if for all $f : \mathbf{s}_1 \times \cdots \times \mathbf{s}_n \rightarrow \mathbf{s}'$ and all $t_1,\ldots,t_n \in T_\Sigma$ where t_i is of sort $\mathbf{S}_i$

$$\mathbf{D}(f^A({t_1}^A,\ldots,{t_n}^A)) \Rightarrow \phi_\mathbf{s}(f^A({t_1}^A,\ldots,{t_n}^A)) = f^B(\phi_{\mathbf{s}_1}({t_1}^A),\ldots,\phi_{\mathbf{s}_n}({t_n}^A))$$

This is the homomorphism condition commonly used in the literature on partial algebras. For our purposes it has to be strengthened: A homomorphism ϕ is *sufficiently defined* if

$$SP \vdash \mathbf{D}(f(t_1,\ldots,t_n)) \Rightarrow B \models \mathbf{D}(\phi_\mathbf{s}(f^A({t_1}^A,\ldots,{t_n}^A)))$$

The definedness predicate has been extended here, to make it applicable to homomorphisms. The notion of sufficiently defined homomorphisms is comparable to

that of total homomorphisms in [1]. The difference is that we only require the homomorphism to be defined if the term $f(\vec{t})$ can be proved to be defined from the specification, whereas a total homomorphism must be defined if $f(\vec{t})$ is defined in A. This implies that every total homomorphism is sufficiently defined, but not the other way round. The two coincide in total algebras and minimally defined algebras (defined later).

The category of all partial algebras over a signature Σ with homomorphisms as morphisms is $\mathsf{PAlg}(\Sigma)$. In a *generated* algebra A , every element in the carriers of A is the interpretation of some ground term. The category $\mathsf{PGAlg}(\Sigma)$ of partial generated Σ-algebras is a subcategory of $\mathsf{PAlg}(\Sigma)$. Another subcategory of $\mathsf{PAlg}(\Sigma)$ is the category of minimally defined partial Σ-algebras:

$$\mathsf{MDPAlg}(\Sigma) = \left\{A \in \mathsf{PAlg}(\Sigma) | \forall t \in \mathrm{T}_\Sigma \,.\, \mathbf{D}(t^{\mathrm{A}}) \Rightarrow \forall B \in \mathsf{PAlg}(\Sigma) \,.\, \mathbf{D}(t^{\mathrm{B}})\right\}$$

A specification may be designated as *primitive*, if it is monomorphic, i.e. all its models are equal up to isomorphism. For our purposes, sorts in primitive specifications are important for their *observability*: values of primitive sort can be observed from outside the algebraic model, because they can be transformed into an effect on the material world, like moving a pointer over a dial or putting on a dot on a fluorescent screen. Values of non-primitive sort can only be observed by applying functions to them, thus mapping them onto values of primitive sorts. We consider only models with non-trivial primitive carrier sets, i.e. primitive carrier sets must have at least two elements. Trivial carrier sets are not interesting, because a computer which can be in only one state is of no use.

A specification SP is built upon the primitive specification PR, if

$Sig[\![\mathrm{PR}]\!] \subseteq Sig[\![\mathrm{SP}]\!]$ (SP extends PR) and
$\forall A \in Mod[\![\mathrm{SP}]\!] \,.\, A|_{Sig[\![\mathrm{PR}]\!]} \in Mod[\![\mathrm{PR}]\!]$ (SP does not introduce junk or confusion)

Here $A|_{Sig[\![\mathrm{PR}]\!]}$ is the $Sig[\![\mathrm{PR}]\!]$-reduct of A (cf. [6]). A term $t \in \mathrm{T}_{Sig[\![\mathrm{SP}]\!]}$ is called primitive if it is of sort $\mathbf{p}$, and $\mathbf{p} \in Sig[\![\mathrm{PR}]\!]_{\mathbf{sorts}}$. The 'no confusion' condition requires that SP does not introduce equalities between primitive terms, that were not implied by PR. Extra elements, which are not part of models of PR are called junk, and may not be introduced (this is sometimes called sufficient completeness).

Example 1. The following is a loose specification of finite sets of natural numbers:

$$\begin{array}{rrl} \text{SETS} = & \textbf{primitive} & \text{BOOL}, \text{NAT} \\ & \textbf{sorts} & \textbf{set} \\ & \textbf{opns} & \emptyset :\rightarrow \textbf{set} \\ & & \{_\} : \textbf{nat} \rightarrow \textbf{set} \\ & & \cup : \textbf{set}, \textbf{set} \rightarrow \textbf{set} \\ & & \in : \textbf{nat}, \textbf{set} \rightarrow \textbf{bool} \\ & & \mathit{empty} : \textbf{set} \rightarrow \textbf{bool} \\ & \textbf{laws} & n \in \emptyset = \mathit{false} \\ & & n \in \{m\} = (n == m) \\ & & n \in (s_1 \cup s_2) = (n \in s_1) \vee (n \in s_2) \\ & & \mathit{empty}(\emptyset) = \mathit{true} \\ & & \mathit{empty}(\{n\}) = \mathit{false} \\ & & \mathit{empty}(s_1 \cup s_2) = \mathit{empty}(s_1) \wedge \mathit{empty}(s_2) \end{array}$$

Note that the specified laws are not sufficient to prove that e.g.

$$s_1 \cup s_2 = s_2 \cup s_1$$

although it is impossible to distinguish between these terms using the operators in $\mathit{Sig}[\![\text{SETS}]\!]_{\textbf{opns}}$. The initial model of SETS only satisfies the laws that are given in the specification. In the category of SETS-models $\mathit{Mod}[\![\text{SETS}]\!]$ there exists a final model which satisfies associativity, commutativity and idempotence. Apart from the initial and final models there are others, which satisfy some, but not all of these properties.

Note that in this case, $\mathsf{PGAlg}(\mathit{Sig}[\![\text{SETS}]\!]) = \mathsf{MDPAlg}(\mathit{Sig}[\![\text{SETS}]\!])$, because there are no undefined terms in any model. This is no longer the case if we add the operation

$$\begin{array}{c} \mathit{extract} : \textbf{set} \rightarrow \textbf{nat} \\ \neg \mathit{empty}(s) \Rightarrow \mathit{extract}(s) \in s \end{array}$$

which yields an arbitrary element from its argument set. Now the term $\mathit{extract}(\emptyset)$ is not defined. Various models may yield an arbitrary integer, or an error when asked to extract from the empty set.

The SETS specification is built on the primitive specifications BOOL and NAT because it does not imply equations like $\mathit{true} = \mathit{false}$ or $\mathit{succ}(0) = 0$, that do not hold in models of the primitive specification (no confusion). It is also sufficiently complete with respect to both primitive specification, because no 'extra' (junk) elements of sorts **bool** or **nat** are introduced.

□

In order to be able to reason about a carrier set element as the result of some computation, we need a notion of 'program'. A program corresponds to an 'accumulated arrow' as defined in [3], but we shall stay more closely to the familiar concept of terms with variables.

First of all, for every sort $\mathbf{s}$ in the signature Σ the identity operation on $\mathbf{s}$, denoted by $\mathrm{Id}_{\mathbf{s}} : s \rightarrow s$ is introduced. For every term $t \in \mathrm{T}_{\Sigma}$ of sort $\mathbf{s}$, $\mathrm{Id}_{\mathbf{s}}(t)$ is a term, and $\mathrm{Id}_{\mathbf{s}}(t) = t$ always holds. Frequently we shall drop the subscript and write $\mathrm{Id}(t)$.

A program is a term with variables,

$$t \in \mathrm{T}_{\Sigma}(x_1, \ldots, x_n)$$

If t is interpreted in an algebra A and applied to the 'input data' $(u_1, \ldots, u_n) \in |A|^*$, the 'computation' $t^{\mathrm{A}}[u_1/x_1, \ldots, u_n/x_n]$ will result. We can define an ordering on the variables $x_1, \ldots, x_n$ and treat them as parameters of the function t; Instead of

$$\lambda u_1, \ldots, u_n \,.\, t[u_1/x_1, \ldots, u_n/x_n]$$

we shall just write t. Similarly, $t(u_1, \ldots, u_n)$ is a shorthand for

$$t[u_1/x_1, \ldots, u_n/x_n]$$

We shall use the following notations for groups:

$$\vec{x} \text{ stands for } (x_1, \ldots, x_n)$$
$$x^* \text{ stands for } \underbrace{(x, \ldots, x)}_{n}$$

In both cases n is determined by the fact that expressions must be correctly typed. We shall denote programs by capital letters (e.g. $F \in \mathrm{T}_{\Sigma}(\vec{x}), \vec{G} \in \mathrm{T}_{\Sigma}(\vec{x})^*$) to distinguish them from operation symbols (e.g. $f \in \Sigma_{\mathbf{opns}}$).

The form of programs as terms is not convenient for our purposes; we often want to 'cut out' one or more subexpressions and seperate them from the rest of the program. This is facilitated by the following constructions:

- Identity introduction: for all $t_1, \ldots, t_n, t_i \in \mathrm{T}_{\Sigma}$ of sort $\mathbf{s}_i$:

  $$(t_1, \ldots, t_n) = (\mathrm{Id}_{\mathbf{s}_1}(t_1), \ldots, \mathrm{Id}_{\mathbf{s}_n}(t_n)) = \mathrm{Id}^*(\vec{t})$$

  A special case is the 'nullary' identity Id^0 with no parameters and no result, e.g. $(1 + 2)(\mathrm{Id}^0) = (1 + 2)$.

- Composition of programs is expressed by the ';' symbol:

  $$(\vec{F}; \vec{G})(\vec{t}) \stackrel{\mathrm{def}}{=} \vec{G}(\vec{F}(\vec{t}))$$

  This is associative, i.e.

  $$(\vec{F}; \vec{G}); \vec{H} = \vec{F}; (\vec{G}; \vec{H}) = \vec{F}; \vec{G}; \vec{H}$$

  For example $(+; sqrt)(2, 2) = sqrt(2 + 2)$.

- Functions can be grouped with the ',' symbol:

$$(\vec{F},\vec{G})(\vec{t_1},\vec{t_2}) = (\vec{F}(\vec{t_1}),\vec{G}(\vec{t_2}))$$

  This is also associative. Note that both sides must be correctly typed; There is at most one way to split the group of terms $\vec{t} = (\vec{t_1},\vec{t_2})$ so that the above is correct. For example $((+,-);*)(4,3,2,1) = (4+3)*(2-1)$.

The following lemma is useful for cutting out subexpressions:

Lemma 1 : cutting out subexpressions
For all $\vec{F_1},\vec{F_2},\vec{G_1},\vec{G_2} \in \mathrm{T}_\Sigma(\vec{x})^*$,

$$((\vec{F_1};\vec{F_2}),(\vec{G_1};\vec{G_2})) = ((\vec{F_1},\vec{G_1});(\vec{F_2},\vec{G_2}))$$

provided that both sides are correctly typed.

Proof:

$$\begin{array}{rcl}
((\vec{F_1};\vec{F_2}),(\vec{G_1};\vec{G_2}))(\vec{t_1},\vec{t_2}) & = & \text{(def. of ,)} \\
((\vec{F_1};\vec{F_2})(\vec{t_1}),(\vec{G_1};\vec{G_2})(\vec{t_2})) & = & \text{(def. of ;)} \\
(\vec{F_2}(\vec{F_1}(\vec{t_1})),\vec{G_2}(\vec{G_1}(\vec{t_2}))) & = & \text{(def. of ,)} \\
(\vec{F_2},\vec{G_2})(\vec{F_1}(\vec{t_1}),(\vec{G_1}(\vec{t_2})) & = & \text{(def. of ,)} \\
(\vec{F_2},\vec{G_2})((\vec{F_1},\vec{G_1})(\vec{t_1},\vec{t_2})) & = & \text{(def. of ;)} \\
((\vec{F_1},\vec{G_1});(\vec{F_2},\vec{G_2}))(\vec{t_1},\vec{t_2}) & &
\end{array}$$

□

The following example shows how the new constructions and the cutting lemma may be used to separate subexpressions from the rest of a program.

Example 2. Consider the expression

$$(\underline{(1+2)}-3)*\underline{(4+5)}$$

We want to seperate the underlined subexpressions from the rest of the program. This can be done by rewriting the expression:

$$\begin{array}{lcl}
((1+2)-3)*(4+5) & = & \text{(rewrite using , and ;)} \\
((((1+2),3);-),(4+5));* & = & \text{(Id-introduction)} \\
(((((1+2);\mathrm{Id}),(\mathrm{Id}^0;3));-),((4+5);\mathrm{Id}));* & = & \text{(cutting lemma)} \\
(((((1+2),\mathrm{Id}^0);(\mathrm{Id},3));-),((4+5);\mathrm{Id}));* & = & \text{(cutting lemma)} \\
((((1+2);(\mathrm{Id},3)),(4+5));(-,\mathrm{Id}));* & = & \text{(Id-introduction)} \\
(((1+2);(\mathrm{Id},3)),((4+5);\mathrm{Id}));(-,\mathrm{Id});* & = & \text{(cutting lemma)} \\
(((1+2),(4+5));((\mathrm{Id},3),\mathrm{Id}));(-,\mathrm{Id});* & = & \\
(\underline{(1+2)},\underline{(4+5)});(\mathrm{Id},3,\mathrm{Id});(-,\mathrm{Id});* & &
\end{array}$$

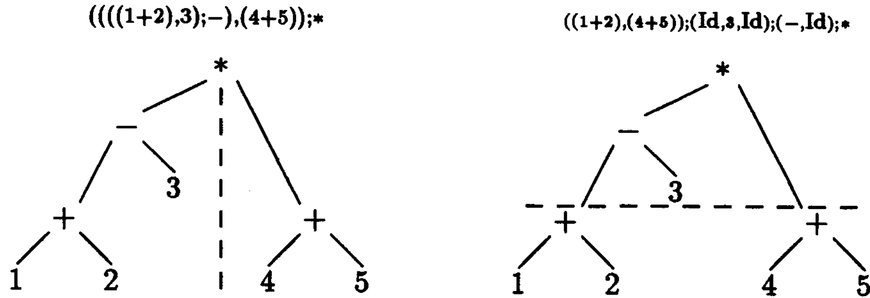

Figure 2: Dividing the expression tree

In the last line the underlined subexpressions are neatly separated from the rest of the expression. The nullary identity operator Id^0 was used in the third and fourth lines as a placeholder to expose the use of the cutting lemma more clearly. It is, however, not necessary, and we could as well have the empty string substituted for one of the $\vec{F}_i$ or $\vec{G}_i$ in the cutting lemma.

The cutting out of subexpressions can be looked upon as dividing the expression-tree: in this example we wanted the tree divided horizontally, instead of vertically (see figure 2). In this way we can split programs, compute everything below the horizontal line, do something with the intermediate results (converting them, for instance), and then continue with the rest of the program above the horizontal line.

□

3 Data conversions between different models

Let SP be a specification built upon a primitive specification PR with signature $\Pi = \mathit{Sig}[\![\mathrm{PR}]\!]$. We have $\Sigma = \mathit{Sig}[\![\mathrm{SP}]\!]$, and $\mathcal{M} = \mathit{Mod}[\![\mathrm{SP}]\!]$. In order to convert data from the model $A \in \mathcal{M}$ to another model $B \in \mathcal{M}$, we need a $\Sigma_{\mathbf{sorts}}$-sorted mapping

$$\Gamma_{\mathrm{A,B}} : A \rightarrow B = \left\{ (\Gamma_{\mathrm{A,B}})_{\mathbf{s}} : \mathbf{s}^{\mathrm{A}} \rightarrow \mathbf{s}^{\mathrm{B}} \right\}$$

satisfying requirements such that we feel that it correctly converts data from $|A|$ to $|B|$. For brevity, we shall mention the fact that mappings are $\Sigma_{\mathbf{sorts}}$-sorted no more, and drop the subscript $\mathbf{s}$.

Only a small fraction of all possible mappings $\Gamma_{\mathrm{A,B}} : A \rightarrow B$ can be correct data conversions, and our task is to find a condition against which to test these mappings. Intuitively, a mapping $\Gamma_{\mathrm{A,B}}$ is a correct data conversion if it preserves

the meaning of every object in $|A|$: At any point in a computation, we may interrupt the computation, convert the intermediate results to another model, and continue the computation there, without changing the meaning of the final result.

To formalize the notion of 'meaning', we have to realize that elements of carrier sets are the 'internal representations' of data. They only become meaningful when * someone extracts information about them, by feeding them to programs with output in primitive sorts. Therefore the meaning of some element of a carrier set determines the result of the application of a primitive compound function to this element. This result can be represented by a primitive ground term, which is the 'external representation' of the corresponding element of a primitive carrier set.

The meaning of $a \in |A|$ is given by the function

$$\begin{aligned} meaning^{\mathbf{A}}(a) = \{\langle F, v\rangle \mid\ & F \in \mathrm{T}_\Sigma(x)\,|_\Pi, \\ & v \in \mathrm{T}_\Pi, \\ & \exists t \in \mathrm{T}_\Sigma \,.\, (\ t^{\mathbf{A}} = a \wedge \mathrm{SP} \vdash \mathbf{D}(F(t))\ ), \\ & F^{\mathbf{A}}(a) = v^{\mathbf{A}}\} \end{aligned}$$

The term algebra $\mathrm{T}_\Sigma(x)\,|_\Pi$ contains exactly the primitive terms in $\mathrm{T}_\Sigma(x)$). The compound function F is a 'test function' with primitive result, and v is its result when applied to a. The third line expresses that only 'legal' programs, that can be proved to be correct, are considered.

A data conversion $\Gamma_{\mathbf{A},\mathbf{B}} : A \to B$ is not necessarily a total mapping. There may be elements in $|A|$ that will never occur as the interpretation of a ground term (which models a possible computation). These elements are not term-generated, and we do not expect $\Gamma_{\mathbf{A},\mathbf{B}}$ to convert them. Also, if A is not minimally defined, there may be elements in $|A|$ that do occur as the interpretation of some $t \in \mathrm{T}_\Sigma$, but for which there is no such t that can be proved to be defined; $\mathrm{SP} \not\vdash \mathbf{D}(t)$ whereas $A \models \mathbf{D}(t)$. We do not expect $\Gamma_{\mathbf{A},\mathbf{B}}$ to convert these either.

Example 3. Consider the following extension of the natural numbers:

$$\begin{aligned} \mathrm{XNAT} = \ & \textbf{primitive}\ \mathrm{NAT} \\ & \qquad\quad \textbf{opns}\ \ sqrt : \textbf{nat} \to \textbf{nat} \\ & \qquad\quad \textbf{laws}\ \ sqrt(n * n) = n \end{aligned}$$

The following are two implementations of XNAT:

$$\begin{aligned} \mathrm{XNAT1} = \ & \textbf{primitive}\ \ \mathrm{BOOL}, \mathrm{NAT} \\ & \qquad\ \ \textbf{opns}\ \ sqrt : \textbf{nat} \to \textbf{nat} \\ & \qquad\qquad\quad\ \ try : \textbf{nat}, \textbf{nat} \to \textbf{nat} \\ & \qquad\ \ \textbf{laws}\ \ sqrt(n) = try(n, 0) \\ & \qquad\qquad\quad\ \ m * m = n \Rightarrow try(n, m) = m \\ & \qquad\qquad\quad\ \ m * m \neq n \Rightarrow try(n, m) = try(n, m + 1) \end{aligned}$$

$$
\begin{array}{lll}
\text{XNAT2} = & \textbf{primitive} & \text{NAT} \\
& \textbf{opns} & sqrt : \textbf{nat} \to \textbf{nat} \\
& & try : \textbf{nat}, \textbf{nat} \to \textbf{nat} \\
& \textbf{laws} & sqrt(n) = try(n, n) \\
& & m * m > n \Rightarrow try(n, m) = try(n, m - 1) \\
& & m * m \le n \wedge (m + 1) * (m + 1) > n \Rightarrow try(n, m) = m
\end{array}
$$

Let $A \in Mod[\![\text{XNAT1}]\!]$ be minimally defined, $B \in Mod[\![\text{XNAT2}]\!]$. Now XNAT $\nvdash$ $\mathbf{D}(sqrt(5))$, and $A \not\models \mathbf{D}(sqrt(5))$ whereas $B \models \mathbf{D}(sqrt(5))$. The term $sqrt(5)$ cannot be proved to be defined from XNAT, and we do not require that $\Gamma_{A,B} : A \to B$ can convert it.

□

The above 'definedness properties' must also hold for the objects in $|B|$ obtained by applying $\Gamma_{A,B}$, if we want to be able to compose data conversions. We shall see why this is so in proposition 4.

The following definition summarizes all the above requirements in a formal way:

Definition 1 : correctness of a data conversion

Let $\text{SP}, \Sigma, \mathcal{M}$ be defined as before, and $A, B \in \mathcal{M}$.

A mapping $\Gamma_{A,B} : A \to B$ is a correct data conversion if

$$
\begin{array}{l}
\text{for all } \vec{t} \in \text{T}_\Sigma^*, \vec{F} \in \text{T}_\Sigma(\vec{x})^* \text{ and primitive } G \in \text{T}_\Sigma(\vec{y}) \\
\text{if SP} \vdash \mathbf{D}(\ (\vec{F}; G)(\vec{t})\ ) \\
\text{then } (\vec{F}^A; \Gamma_{A,B}^*; G^B)(\vec{t}^A) = (\Gamma_{A,B}^*; \vec{F}^B; G^B)(\vec{t}^A) \\
\qquad \text{and there exist } \vec{t'}, \vec{t''} \in \text{T}_\Sigma^* \\
\qquad \text{such that} \\
\qquad \vec{t'}^B = \Gamma_{A,B}^*(\vec{t}^A), \quad \text{SP} \vdash \mathbf{D}(\ (\vec{F}; G)(\vec{t'})\ ), \\
\qquad \vec{t''}^B = (\vec{F}^A; \Gamma_{A,B}^*)(\vec{t}^A), \quad \text{SP} \vdash \mathbf{D}(\ G(\vec{t''})\ )
\end{array}
$$

A graphical representation of this correctness condition is given in figure 3. Note how the program is 'cut' between $\vec{F}$ and G, in order to convert the intermediate results computed by applying $\vec{F}^A$ to the input $\vec{t}^A$. This can be done with any program, using lemma 1.

A special case of the correctness condition is expressed in the following

Fact 1 :

Let $\Pi = Sig[\![\text{PR}]\!]$. If we take $\vec{F} = p$, $p \in \text{T}_\Pi$ a primitive ground term, and $G = \text{Id}$, the identity function, then the correctness condition implies

$$\Gamma_{A,B}(p^A) = p^B$$

For instance, if PR = BOOL, the specification of the boolean data type, this implies

$$\Gamma_{A,B}(true^A) = true^B$$

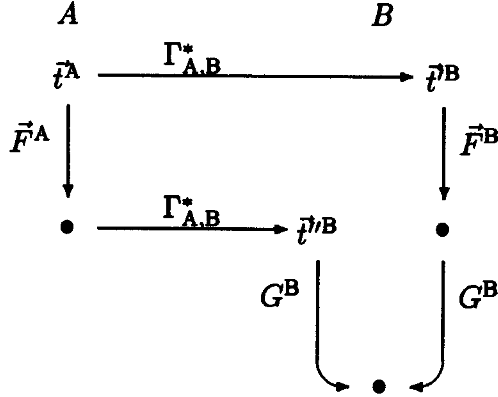

Figure 3: Correctness of a data conversion

and

$$\Gamma_{A,B}(false^{A}) = false^{B}$$

Example 4. Recall example 3. In the correctness condition take

$$\vec{t} = 5, \vec{F} = sqrt, G = \lambda x \,.\, x == 2$$

(where == is the equality operation in NAT) in proposition 4, and consider what would happen if we had not required XNAT ⊢ **D**($sqrt(5) == 2$) to hold if $\Gamma_{A,B}$ is to convert this. Then the correctness condition would require that

$$\begin{aligned} &true^{B} = \\ &(sqrt^{B}(5^{B}) == {}^{B}2^{B}) = \\ &(sqrt^{B}(\Gamma_{A,B}(5^{A})) == {}^{B}2^{B}) = \\ &(\Gamma_{A,B}(sqrt^{A}(5^{A})) == {}^{B}2^{B}) \end{aligned}$$

but the last term is undefined, since $sqrt^{A}(5^{A})$ is undefined and $==^{B}$ is strict.
□

Proposition 1 : data conversions preserve meaning
If $\Gamma_{A,B} : A \to B$ is a correct data conversion then

*

$$\forall t \in T_{\Sigma} \,.\, meaning^{A}(t^{A}) \subseteq meaning^{B}(\Gamma_{A,B}(t^{A}))$$

Proof: Let $\langle F, v\rangle \in \mathit{meaning}^{\mathrm{A}}(t^{\mathrm{A}})$ Now correctness of $\Gamma_{\mathrm{A,B}}$ implies that there exists a term $t' \in \mathrm{T}_\Sigma$ such that

$$t'^{\mathrm{B}} = \Gamma_{\mathrm{A,B}}(t^{\mathrm{A}}) \wedge \mathrm{SP} \vdash \mathbf{D}(F(t))$$

Also,

$$\begin{array}{lll} F^{\mathrm{B}}(\Gamma_{\mathrm{A,B}}(t^{\mathrm{A}})) & = & \\ (\Gamma_{\mathrm{A,B}}; F^{\mathrm{B}}; \mathrm{Id}^{\mathrm{B}})(t^{\mathrm{A}}) & = & (\text{correctness of } \Gamma_{\mathrm{A,B}}) \\ (F^{\mathrm{A}}; \Gamma_{\mathrm{A,B}}; \mathrm{Id}^{\mathrm{B}})(t^{\mathrm{A}}) & = & (\langle F, v\rangle \in \mathit{meaning}^{\mathrm{A}}(t^{\mathrm{A}})) \\ \Gamma_{\mathrm{A,B}}(v^{\mathrm{A}}) & = & (\text{correctness of } \Gamma_{\mathrm{A,B}} \text{ and fact 1}) \\ v^{\mathrm{B}} & & \end{array}$$

so $\langle F, v\rangle \in \mathit{meaning}^{\mathrm{B}}(\Gamma_{\mathrm{A,B}}(t^{\mathrm{A}}))$.

□

It may seem surprising that $\mathit{meaning}^{\mathrm{A}}(t^{\mathrm{A}})$ is a subset of $\mathit{meaning}^{\mathrm{B}}(\Gamma_{\mathrm{A,B}}(t^{\mathrm{A}}))$, instead of the two sets being equal. Indeed the image under data conversion may be 'better defined' than the original. The important thing, however, is that for all programs F such that $F(t)$ can be proved to be defined, there is a tuple $\langle F, v\rangle \in \mathit{meaning}^{\mathrm{A}}(t^{\mathrm{A}})$. For all tuples $\langle G, v\rangle \in \mathit{meaning}^{\mathrm{B}}(\Gamma_{\mathrm{A,B}}(t^{\mathrm{A}}))$ that are not in $\mathit{meaning}^{\mathrm{A}}(t^{\mathrm{A}})$, we have $\mathrm{SP} \not\vdash \mathbf{D}(G(t))$. The set $\mathit{meaning}^{\mathrm{A}}(t^{\mathrm{A}})$ cannot 'shrink', because of the condition $\mathrm{SP} \vdash \mathbf{D}(F(t))$ in the definition of *meaning*.

Example 5. In example 4 the tuple

$$\langle (\lambda x \,.\, \mathit{sqrt}(x) == 2), \mathit{true}\rangle$$

is not in $\mathit{meaning}^{\mathrm{A}}(5^{\mathrm{A}})$, but it *is* in $\mathit{meaning}^{\mathrm{B}}(\Gamma_{\mathrm{A,B}}(5^{\mathrm{A}}))$.

□

In practice, proving whether a mapping $\Gamma_{\mathrm{A,B}} : A \to B$ satisfies the correctness condition may turn out to be very difficult. The following proposition provides a simpler version of the correctness condition which is, in some cases, easier to prove. The essence of the proposition is that it is not necessary to consider every possible program $\vec{F} \in \mathrm{T}_\Sigma(\vec{x})$ when proving $\Gamma_{\mathrm{A,B}}$ correct, but only simple fuctions $f \in \Sigma_{\mathrm{opns}}$. *

Proposition 2 : simplifying the correctness condition
The correctness condition can be simplified without changing its meaning, by replacing $\vec{F} \in \mathrm{T}_\Sigma(\vec{x})^*$ by $f \in \Sigma_{\mathrm{opns}}$. It then turns into:

$$\begin{array}{l} \text{for all } \vec{t} \in \mathrm{T}^*_\Sigma, f \in \Sigma_{\mathrm{opns}}, \text{ and primitive } G \in \mathrm{T}_\Sigma(y) \\ \text{there exist } \vec{t'} \in \mathrm{T}^*_\Sigma, t'' \in \mathrm{T}_\Sigma \\ \text{such that} \\ \text{if } \mathrm{SP} \vdash \mathbf{D}(\ (f; G)(\vec{t})\ ) \\ \text{then } \vec{t'}^{\mathrm{B}} = \Gamma^*_{\mathrm{A,B}}(\vec{t}^{\mathrm{A}}),\ \ \mathrm{SP} \vdash \mathbf{D}(\ (f; G)(\vec{t'})\ ), \\ \quad t''^{\mathrm{B}} = (f^{\mathrm{A}}; \Gamma^*_{\mathrm{A,B}})(\vec{t}^{\mathrm{A}}),\ \ \mathrm{SP} \vdash \mathbf{D}(\ G(t'')\ ), \\ \quad (f^{\mathrm{A}}; \Gamma^*_{\mathrm{A,B}}; G^{\mathrm{B}})(\vec{t}^{\mathrm{A}}) = (\Gamma^*_{\mathrm{A,B}}; f^{\mathrm{B}}; G^{\mathrm{B}})(\vec{t}^{\mathrm{A}}) \end{array}$$

Proof: Assume that $\Gamma_{A,B} : A \rightarrow B$ has been proved correct for all $f \in \Sigma_{opns}$, i.e. the above variant of the correctness condition holds. We prove that this implies that $\Gamma_{A,B}$ is correct for all $\vec{F} \in T_\Sigma(\vec{x})^*$ and all $G \in T_\Sigma|_\Pi$ by induction on the structure of $\vec{F}$:

- $\vec{F} = f \in \Sigma_{opns}$:
  In this case the proposition obviously holds.

- $\vec{F} = (\vec{F_1}, \vec{F_2}),\ \vec{F_1}, \vec{F_2} \in T_\Sigma(\vec{x})^*$:
  In this case for all $\vec{t} \in T^*_\Sigma$ such that $SP \vdash \mathbf{D}((\vec{F}; G)(\vec{t}))$ we can split

  $$\vec{t} = (\vec{t_1}, \vec{t_2})$$

  (using lemma 1 if necessary) and

  $$\vec{F}(\vec{t}) = (\vec{F_1}, \vec{F_2})(\vec{t_1}, \vec{t_2})$$

  The induction hypothesis is that $\Gamma_{A,B}$ is correct for $\vec{F_1}$:

  $$\begin{array}{l}(\vec{F_1}^A; \Gamma^*_{A,B};\ ((\mathrm{Id}^*, (\vec{F_2}^A; \Gamma^*_{A,B})(t_2{}^A)); G^B)\ )(\vec{t_1}^A) = \\ (\Gamma^*_{A,B}; \vec{F_1}^B;\ ((\mathrm{Id}^*, (\vec{F_2}^A; \Gamma^*_{A,B})(t_2{}^A)); G^B)\ )(\vec{t_1}^A)\end{array}$$

  And similarly for $\vec{F_2}$. With a few applications of the cutting lemma, one can derive

  $$((\vec{F_1}^A, \vec{F_2}^A); \Gamma^*_{A,B}; G^B)(\vec{t}^A) = (\Gamma^*_{A,B}; (\vec{F_1}^B, \vec{F_2}^B); G^B)(\vec{t}^A)$$

  Together with the definedness conditions, which are easily proved, this shows that $\Gamma_{A,B}$ is correct for $\vec{F}$.

- $\vec{F} = (\vec{F_1}; \vec{F_2}), \vec{F_1}, \vec{F_2} \in T_\Sigma(\vec{x})^*$:
  Now for all $\vec{t} \in T^*_\Sigma$ such that $SP \vdash \mathbf{D}((\vec{F_1}; \vec{F_2}; G)(\vec{t}))$ our induction hypothesis is

  $$\begin{array}{ll}\exists \vec{t_1}, \vec{t_1}'. & \vec{t_1}^B = \Gamma^*_{A,B}(\vec{t}^A),\ \ SP \vdash \mathbf{D}((\vec{F_1}; \vec{F_2}; G)(\vec{t_1})), \\ & \vec{t_1}'^B = (\vec{F_1}^A; \Gamma^*_{A,B})(\vec{t}^A),\ \ SP \vdash \mathbf{D}((\vec{F_2}; G)(\vec{t_1}')), \\ & (\vec{F_1}^A; \Gamma^*_{A,B}; \vec{F_2}^B; G^B)(\vec{t}^A) = (\Gamma^*_{A,B}; \vec{F_1}^B; \vec{F_2}^B; G^B)(\vec{t}^A)\end{array}$$

  and

  $$\begin{array}{ll}\exists \vec{t_2}, \vec{t_2}'. & \vec{t_2}^B = \Gamma^*_{A,B}(\vec{F_1}^A(\vec{t}^A)),\ \ SP \vdash \mathbf{D}((\vec{F_2}; G)(\vec{t_2})), \\ & \vec{t_2}'^B = (\vec{F_2}^A; \Gamma^*_{A,B})(\vec{F_1}^A(\vec{t}^A)),\ \ SP \vdash \mathbf{D}(G(\vec{t_2}')), \\ & (\vec{F_2}^A; \Gamma^*_{A,B}; G^B)(\vec{F_1}^A(\vec{t}^A)) = (\Gamma^*_{A,B}; \vec{F_2}^B; G^B)(\vec{F_1}^A(\vec{t}^A))\end{array}$$

  With some rewriting and putting $\vec{t_1}' = \vec{t_2}$ one can immediately derive correctness of $\Gamma_{A,B}$ for $\vec{F}$.

□

The correctness condition expresses that we are only interested in the *meaning* of intermediate results, not in particular ways of obtaining these results. This is not immediately clear from definition 1; it is made explicit by the following fact, which shows that some observable result may be 'computed' by different programs in different algebra's, with data conversions between them applied at intermediate stages.

Fact 2 :
If $\Gamma_{A,B} : A \rightarrow B$ is a correct data conversion, then for all $\vec{t} \in T^*_\Sigma$, $\vec{F_1}, \vec{F_2} \in T_\Sigma(\vec{x})^*$ and $G_1, G_2 \in T_\Sigma(y)\,|_\Pi$:

$$\text{If } (\vec{F_1}^A; G_1{}^A)(\vec{t}^A) = (\vec{F_2}^A; G_2{}^A)(\vec{t}^A)$$
$$\text{then } (\vec{F_1}^A; \Gamma^*_{A,B}; G_1{}^B)(\vec{t}^A) = (\Gamma^*_{A,B}; \vec{F_2}^B; G_2{}^B)(\vec{t}^A)$$

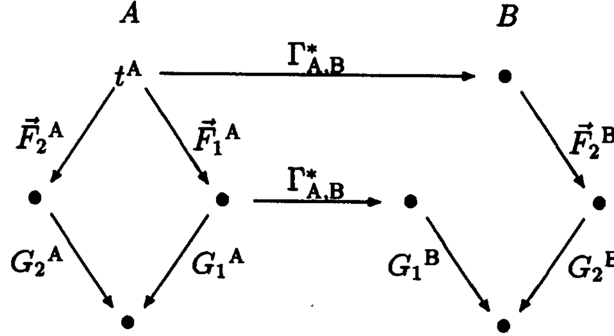

Figure 4: Fact 2

Proof:

$$\begin{aligned}
(\vec{F_1}^A; \Gamma^*_{A,B}; G_1{}^B)(\vec{t}^A) &= \text{(correctness of } \Gamma_{A,B})\\
(\vec{F_1}^A; G_1{}^A; \Gamma^*_{A,B})(\vec{t}^A) &= \text{(apply the premiss)}\\
(\vec{F_2}^A; G_2{}^A; \Gamma^*_{A,B})(\vec{t}^A) &= \text{(correctness of } \Gamma_{A,B})\\
(\vec{F_2}^A; \Gamma^*_{A,B}; G_2{}^B)(\vec{t}^A) &= \text{(correctness of } \Gamma_{A,B})\\
(\Gamma^*_{A,B}; \vec{F_2}^B; G_2{}^B)(\vec{t}^A)
\end{aligned}$$

□

Proposition 3 : the identity is a data conversion
For all $A \in \mathcal{M}$ the identity $\mathrm{Id}^A : A \rightarrow A$ is a correct data conversion.

Proof: If $\text{SP} \vdash \mathbf{D}((\vec{F}; G)(\vec{t}))$, then obviously

$$(\vec{F}^{\text{A}}; \text{Id}^{\text{A}}; G^{\text{A}})(\vec{t}^{\text{A}}) = (\text{Id}^{\text{A}}; \vec{F}^{\text{A}}; G^{\text{A}})(\vec{t}^{\text{A}})$$

holds. The definedness conditions pass into

$$\text{SP} \vdash \mathbf{D}((\vec{F}; G)(\vec{t}))$$

which obviously holds.

□

Proposition 4 : data conversions can be composed
Let $A, B, C \in \mathcal{M}$, and $\Gamma_{\text{A,B}} : A \to B$, $\Gamma_{\text{B,C}} : B \to C$ be correct data conversions. Then the composition of $\Gamma_{\text{A,B}}$ and $\Gamma_{\text{B,C}}$, $\Gamma_{\text{A,C}} = \Gamma_{\text{A,B}}; \Gamma_{\text{B,C}}$ is a correct data conversion.

Proof: We shall first prove that

$$\begin{array}{l}
\text{If } (\vec{F}^{\text{A}}; \Gamma^*_{\text{A,B}}; G^{\text{B}})(\vec{t}^{\text{A}}) = (\Gamma^*_{\text{A,B}}; \vec{F}^{\text{B}}; G^{\text{B}})(\vec{t}^{\text{A}}) \\
\text{and } (\vec{F}^{\text{B}}; \Gamma^*_{\text{B,C}}; G^{\text{C}})(\vec{t}^{\text{C}}) = (\Gamma^*_{\text{B,C}}; \vec{F}^{\text{C}}; G^{\text{C}})(\vec{t}^{\text{C}}) \\
\text{then } (\vec{F}^{\text{A}}; \Gamma^*_{\text{A,B}}; \Gamma^*_{\text{B,C}}; G^{\text{C}})(\vec{t}^{\text{A}}) = (\Gamma^*_{\text{A,B}}; \Gamma^*_{\text{B,C}}; \vec{F}^{\text{C}}; G^{\text{C}})(\vec{t}^{\text{A}})
\end{array}$$

assuming that $\Gamma_{\text{A,B}}$ and $\Gamma_{\text{B,C}}$ are correct data conversions:

$$\begin{array}{ll}
(\vec{F}^{\text{A}}; \Gamma^*_{\text{A,B}}; \Gamma^*_{\text{B,C}}; G^{\text{C}})(\vec{t}^{\text{A}}) = & (\text{correctness of } \Gamma_{\text{B,C}}) \\
(\vec{F}^{\text{A}}; \Gamma^*_{\text{A,B}}; G^{\text{B}}; \Gamma_{\text{B,C}})(\vec{t}^{\text{A}}) = & (\text{correctness of } \Gamma_{\text{A,B}}) \\
(\Gamma^*_{\text{A,B}}; \vec{F}^{\text{B}}; G^{\text{B}}; \Gamma_{\text{B,C}})(\vec{t}^{\text{A}}) = & (\text{correctness of } \Gamma_{\text{B,C}}) \\
(\Gamma^*_{\text{A,B}}; \vec{F}^{\text{B}}; \Gamma^*_{\text{B,C}}; G^{\text{C}})(\vec{t}^{\text{A}}) = & (\text{correctness of } \Gamma_{\text{B,C}}) \\
(\Gamma^*_{\text{A,B}}; \Gamma^*_{\text{B,C}}; \vec{F}^{\text{C}}; G^{\text{C}})(\vec{t}^{\text{A}}) &
\end{array}$$

To complete the proof, we have to show that application of $\Gamma_{\text{A,B}}; \Gamma_{\text{B,C}}$ to a defined object $t^{\text{A}}, \text{SP} \vdash \mathbf{D}(t)$ results in the interpretation in C of a defined term $\vec{t}'$. Let $\text{SP} \vdash \mathbf{D}((\vec{F}; G)(\vec{t}))$, then because $\Gamma_{\text{A,B}}$ is a correct data conversion:

$$\begin{array}{l}
\text{there exist } \vec{t}', \vec{t}'' \in \text{T}^*_\Sigma \text{ such that} \\
\vec{t}'^{\text{B}} = \Gamma^*_{\text{A,B}}(\vec{t}^{\text{A}}),\ \text{SP} \vdash \mathbf{D}((\vec{F}; G)(\vec{t}')), \\
\vec{t}''^{\text{B}} = (\vec{F}^{\text{A}}; \Gamma^*_{\text{A,B}})(\vec{t}^{\text{A}}),\ \text{SP} \vdash \mathbf{D}(G(\vec{t}''))
\end{array}$$

Now, because $\Gamma_{\text{B,C}}$ is also a correct data conversion:

$$\begin{array}{l}
\text{there exist } \vec{u}', \vec{u}'' \in \text{T}^*_\Sigma \text{ such that} \\
\vec{u}'^{\text{C}} = \Gamma^*_{\text{B,C}}(\vec{t}'^{\text{B}}) = (\Gamma^*_{\text{A,B}}; \Gamma^*_{\text{B,C}})(\vec{t}^{\text{A}}),\ \text{SP} \vdash \mathbf{D}((\vec{F}; G)(\vec{u}')), \\
\vec{u}''^{\text{C}} = (\text{Id}^{\text{B}}; \Gamma^*_{\text{B,C}})(\vec{t}''^{\text{B}}) = (\vec{F}^{\text{A}}; \Gamma^*_{\text{A,B}}; \Gamma^*_{\text{B,C}})(\vec{t}^{\text{A}}),\ \text{SP} \vdash \mathbf{D}(G(\vec{u}''))
\end{array}$$

which proves the definedness part of the correctness condition for $\Gamma_{\text{A,B}}; \Gamma_{\text{B,C}}$.

□

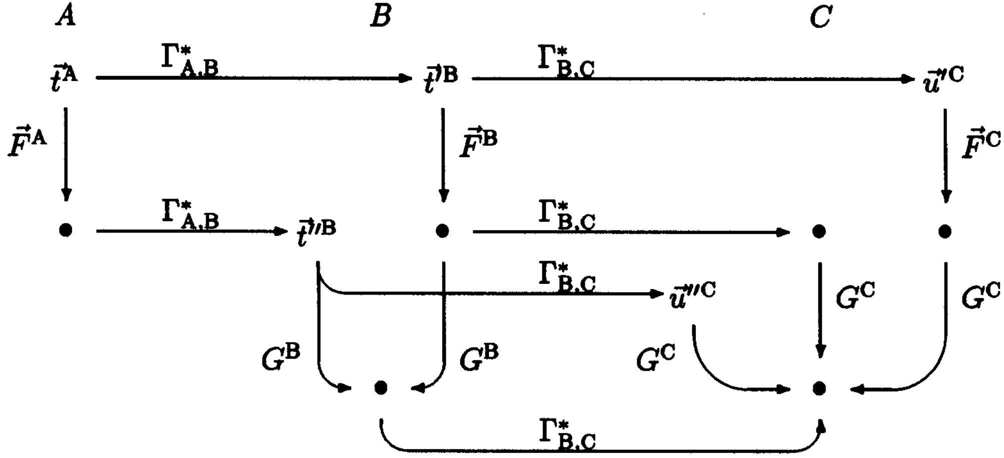

Figure 5: Composition of data conversions

The structure of this proof is depicted in figure 5.

The last part of the proof is especially important, because it shows why the correctness condition requires that provable definedness (SP $\vdash$ **D**(...)) is preserved.

To see this, suppose that B is not generated by provably defined terms, i.e.

$$B \not\models \forall b \in |B| \,.\, \exists t \in \mathrm{T}_\Sigma \,.\, \mathrm{SP} \vdash \mathbf{D}(t) \wedge t^{\mathrm{B}} = b$$

Now some object in $|A|$ might be converted to an $b \in |B|$ which is not term-generated. But then this object b cannot be converted correctly by $\Gamma_{\mathrm{B,C}}$, because it does not satisfy the presumption made by This is illustrated in figure 6.

Since data conversions are special cases of ordinary mappings, composition of data conversions is associative. Thus we conclude from propositions 3 and 4 that *Mod*⟦SP⟧ forms a category, with data conversions as morphisms.

The following proposition shows that the condition for sufficiently defined homomorphisms is stronger than the correctness condition for data conversions. Actually, on primitive sorts sufficiently defined homomorphisms and data conversions coincide (this follows directly from fact 1).

Proposition 5 : sufficiently defined homomorphisms are data conversions

A sufficiently defined homomorphism $\phi : A \rightarrow B$ is a correct data conversion from A to B.

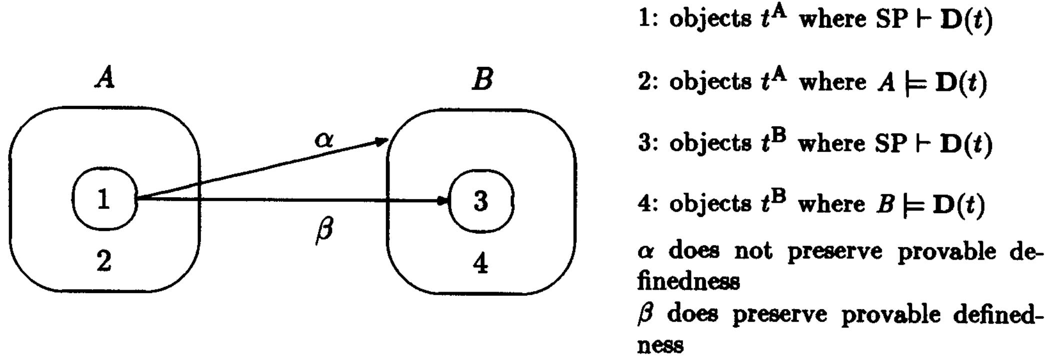

Figure 6: Data conversions must preserve provable definedness

Proof: We prove that the correctness condition (definition 1) holds, with ϕ substituted for $\Gamma_{\mathbf{A},\mathbf{B}}$.

If SP $\vdash \mathbf{D}((\vec{F};G)(\vec{t}))$, then because of the strictness of $\vec{F}$ and G, $\vec{F}(\vec{t})$ and $\vec{t}$ are defined. Now the homomorphism condition for ϕ implies:

$$(\vec{F}^{\mathbf{A}};\phi^*)(\vec{t}^{\mathbf{A}}) = (\phi^*;\vec{F}^{\mathbf{B}})(\vec{t}^{\mathbf{A}})$$

By applying $G^{\mathbf{B}}$ to both sides of this equation we precisely get the last part of the correctness condition:

$$(\vec{F}^{\mathbf{A}};\phi^*;G^{\mathbf{B}})(\vec{t}^{\mathbf{A}}) = (\phi^*;\vec{F}^{\mathbf{B}};G^{\mathbf{B}})(\vec{t}^{\mathbf{A}})$$

A consequence of the condition for suficient definedness is that there exist $\vec{t}', \vec{t}'' \in \mathrm{T}^*_{\Sigma}$, such that

$$\vec{t}'^{\mathbf{B}} = \phi^*(\vec{t}^{\mathbf{A}}) = \vec{t}^{\mathbf{B}}$$
$$\vec{t}''^{\mathbf{B}} = \phi^*(\vec{F}^{\mathbf{A}}(\vec{t}^{\mathbf{A}})) = \vec{F}^{\mathbf{B}}(\phi^*(\vec{t}^{\mathbf{A}})) = \vec{F}^{\mathbf{B}}(\vec{t}^{\mathbf{B}})$$

So if we choose $\vec{t}' = \vec{t}$ and $\vec{t}'' = \vec{F}(\vec{t})$,

$$\text{SP} \vdash \mathbf{D}((\vec{F};G)(\vec{t}')) \text{ passes into SP} \vdash \mathbf{D}((\vec{F};G)(\vec{t}))$$

and

$$\text{SP} \vdash \mathbf{D}(G(\vec{t}'')) \text{ also passes into SP} \vdash \mathbf{D}((\vec{F};G)(\vec{t}))$$

which is the premiss of the correctness condition.

□

4 Abstract data conversions

In the previous section a condition was given under which a particular mapping from one model of a specification to another is a data conversion. But this does not tell us anything about the possibility of converting data from an *arbitrary* model to another. If our theory is going to be useful, it must apply to the model class as a whole; given a specification SP, we would like to know if there exists a data conversion $\Gamma_{\mathrm{A,B}} : A \rightarrow B$ for *every* pair $A, B \in \mathit{Mod}[\![\mathrm{SP}]\!]$. This requires a *syntactic* condition, and the nature of this condition depends on the logical system (institution, [5]) we are working in.

In view of the above, the results in this section are still far from satisfactory. This is partly due to the nature of definition 1, which is really a mixture of a syntactic condition (the terms that must be provably defined by SP) and a semantic condition (the equality between objects in $|B|$).

As a generalization of data conversions, we introduce ***abstract data conversions.***

Definition 2 : abstract data conversion
Let SP be a specification, $\Sigma = \mathit{Sig}[\![\mathrm{SP}]\!]$ and $\mathcal{M} = \mathit{Mod}[\![\mathrm{SP}]\!]$.

An abstract data conversion $\Gamma(\mathcal{M})$ on $\mathcal{M}$ is a family of data conversions:

$$\Gamma(\mathcal{M}) = \{\Gamma_{\mathrm{A,B}} : A \rightarrow B \mid A, B \in \mathcal{M}\}$$
where all $\Gamma_{\mathrm{A,B}}$ satisfy the correctness condition

The idea is that between every pair of models, $\Gamma(\mathcal{M})$ provides a data conversion, so data can always be converted, irrespective of the particular implementation one has. The model class $\mathcal{M}$ is sometimes called the ***abstract data type*** specified by SP, hence the name abstract data conversion. Note that $\Gamma(\mathcal{M})$ is not uniquely defined, since there may be algebras $A, B \in \mathcal{M}$ with more than one mapping $\Gamma_{\mathrm{A,B}} : A \rightarrow B$ that satisfies the correctness condition.

What we have to do now is devise some way of constructing $\Gamma(\mathcal{M})$ for arbitrary model classes. This turns out to be far from simple, and there are many specifications for which no abstract data conversion exists. As a first step towards a theory of abstract data conversions, we here present a necessary condition (proposition 6) and a sufficient condition (proposition 7) for the existence of abstract data conversions. These conditions are rather different, so there is a large collection of model classess 'in between'; we cannot tell whether there exists an abstract data conversion on these model classes from propositions 6 and 7.

Before we go on, we define partial completeness (cf. [1]).

Definition 3 : partial completeness
Let SP be a specification built upon a primitive specification PR, and

$$\Sigma = \mathit{Sig}[\![\mathrm{SP}]\!], \Pi = \mathit{Sig}[\![\mathrm{PR}]\!]$$

SP is partially complete if

$$\begin{array}{l} \forall t \in \mathrm{T}_\Sigma|_\Pi \ . \\ \mathrm{SP} \vdash \mathbf{D}(t) \Rightarrow \exists t' \in \mathrm{T}_\Pi \ . \ \mathrm{SP} \vdash t = t' \end{array}$$

Together with the 'no junk' condition which SP must satisfy this implies

$$\begin{array}{l} \nexists t \in \mathrm{T}_\Sigma|_\Pi \ . \\ \exists t', t'' \in \mathrm{T}_\Pi \ . \\ \exists A, B \in \mathit{Mod}[\![\mathrm{SP}]\!] \ . \\ \mathrm{PR} \vdash t' \neq t'' \wedge t^{\mathrm{A}} = t'^{\mathrm{A}} \wedge t^{\mathrm{B}} = t''^{\mathrm{B}} \end{array}$$

We shall use this as an alternative formulation of partial completeness. In a particular logic system, it is sometimes possible to give simple syntactic conditions for the partial completeness of a specification. This is done in corollary 2 of [1] for partial abstract types. Partial completeness is a necessary condition for the existence of abstract data conversions:

Proposition 6 : existence of an abstract data conversion implies partial completeness
If there exists an abstract data conversion $\Gamma(\mathit{Mod}[\![\mathrm{SP}]\!])$, then SP is partially complete.

Proof: We shall prove that if SP is *not* partially complete, the existence of an abstract data conversion on SP leads to a contradiction.

Suppose SP is not partially complete, and it is built upon PR (it does not introduce junk, neither confusion). Because we do not consider trivial primitive models, the carrier set of a PR-model has at least two elements for every sort, so there must be two models $A, B \in \mathcal{M}$, such that for some primitive term $t \in \mathrm{T}_\Sigma$

$$A \models t = t' \text{ and } B \models t = t''$$

where $t', t'' \in \mathrm{T}_\Pi$ (there exist such t' and t''; no junk is allowed), and

$$\mathit{Mod}[\![\mathrm{PR}]\!] \models t' \neq t''$$

(this follows from our alternative formulation of partial completeness). But suppose $\Gamma_{\mathrm{A,B}} : A \rightarrow B$ is a correct data conversion, then

$$\begin{array}{ll} t'^{\mathrm{B}} = & (\text{correctness of } \Gamma_{\mathrm{A,B}} \text{ w.r.t. ground terms}) \\ \Gamma_{\mathrm{A,B}}(t'^{\mathrm{A}}) = & (A \models t = t') \\ \Gamma_{\mathrm{A,B}}(t^{\mathrm{A}}) = & (\text{correctness of } \Gamma_{\mathrm{A,B}}) \\ t^{\mathrm{B}} = & (B \models t = t'') \\ t''^{\mathrm{B}} & \end{array}$$

This contradicts the assumption that $Mod[\![\mathrm{PR}]\!] \models t' \neq t''$, so SP must be partially complete.

□

The following example illustrates what happens if SP is not partially complete:

Example 6. Let SP be specified by

$$
\begin{array}{rrl}
\mathrm{SP} = & \textbf{primitive} & \mathrm{BOOL} \\
& \textbf{sorts} & \mathbf{s} \\
& \textbf{opns} & a, b, c :\rightarrow \mathbf{s} \\
& & f : \mathbf{s} \rightarrow \textbf{bool} \\
& \textbf{laws} & a \neq c \\
& & (a = b) \vee (b = c) \\
& & f(a) = \mathit{true} \\
& & f(c) = \mathit{false}
\end{array}
$$

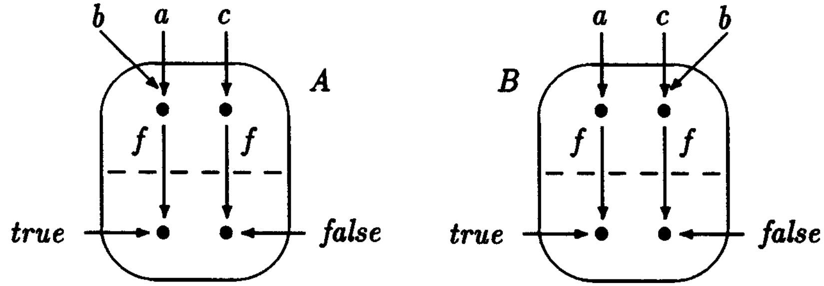

Figure 7: Two models of a specification that is not partially complete

This specification is not partially complete, because $Mod[\![\mathrm{SP}]\!] \models \mathbf{D}(f(b))$ but there is no single $t' \in \mathrm{T}_{Sig[\mathrm{BOOL}]}$ such that $Mod[\![\mathrm{SP}]\!] \models f(b) = t'$. It is not possible to have a correct data conversion $\Gamma_{\mathrm{A,B}} : A \rightarrow B$ between the models A and B as in figure 7. If we (correctly) map a^{A} onto a^{B} and c^{A} onto c^{B}, then

$$(f^{\mathrm{A}}; \Gamma_{\mathrm{A,B}})(b^{\mathrm{A}}) = \mathit{true}^{\mathrm{B}} \neq (\Gamma_{\mathrm{A,B}}; f^{\mathrm{B}})(b^{\mathrm{A}}) = \mathit{false}^{\mathrm{B}}$$

□

A model class $\mathcal{M}$ is called *homomorphic* if there exists a sufficiently defined homomorphism $\phi : A \rightarrow B$ between every $A, B \in \mathcal{M}$. This property is a sufficient condition for the existence of abstract data conversions:

Proposition 7 : abstract data conversions exist for homomorphic model classes
If a model class $\mathcal{M}$ is homomorphic, then there exists an abstract data conversion $\Gamma(\mathcal{M})$ on $\mathcal{M}$.

Proof: For every $A, B \in \mathcal{M}$ there exists a sufficiently defined homomorphism. If we let $\Gamma(\mathcal{M})$ consist of all these homomorphisms, then proposition 5 implies that this is an abstract data conversion.
□

This result may seem to be rather trivial, but it is also very useful, since in many existing systems model classes actually are homomorphic. For example, in the initial algebra semantics abstract data conversions always exist, because all models are isomorphic to the initial model within the class of initial algebras.

5 Implementations and data conversions

We shall briefly review the theory of implementations of algebraic specifications, as developed in [6].

Definition 4 : implementation
Let SP and SP′ be specifications with the same signature. We say that SP′ *is an implementation of* SP, written as

$$\mathrm{SP} \rightsquigarrow \mathrm{SP}'$$

if

$$Mod[\![\mathrm{SP}]\!] \supseteq Mod[\![\mathrm{SP}']\!]$$

Intuitively, this says that the model class becomes smaller, if some implementation detail is specified. Only models in $Mod[\![\mathrm{SP}_0]\!]$ that implement this detail in accordance with the additional information in SP_1 are retained.

Example 7. We can implement sets as specified in example 1 as lists:

$$\text{SETS} \rightsquigarrow \text{LIST-SETS}$$

Lists satisfy associativity, so we have

$$\text{LIST-SETS} \vdash (s_1 \cup s_2) \cup s_3 \ = \ s_1 \cup (s_2 \cup s_3)$$

Now only models in which associativity holds are retained in the model class:

$$\begin{array}{l} Mod[\![\text{SETS}]\!] \not\models (s_1 \cup s_2) \cup s_3 \ = \ s_1 \cup (s_2 \cup s_3) \text{ whereas} \\ Mod[\![\text{LIST-SETS}]\!] \models (s_1 \cup s_2) \cup s_3 \ = \ s_1 \cup (s_2 \cup s_3) \end{array}$$

so $Mod[\![\text{LIST-SETS}]\!] \subseteq Mod[\![\text{SETS}]\!]$.
□

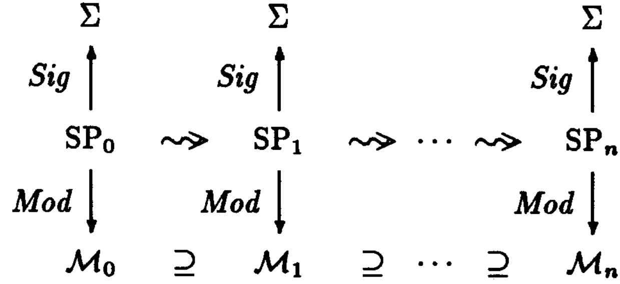

Figure 8: Implementations and model classes

Thus we get a chain of implementations, as in figure 8.
Consider two chains of implementation steps

$$\begin{array}{l} \mathrm{SP}_1 \rightsquigarrow \ldots \rightsquigarrow \mathrm{SP}_n \text{ and} \\ \mathrm{SP}'_1 \rightsquigarrow \ldots \rightsquigarrow \mathrm{SP}'_m \end{array}$$

and the sequences of models of these intermediate implementations,

$$\mathit{Mod}[\![\mathrm{SP}_0]\!] \supseteq \mathit{Mod}[\![\mathrm{SP}_1]\!] \supseteq \cdots \supseteq \mathit{Mod}[\![\mathrm{SP}_n]\!]$$

$$\mathit{Mod}[\![\mathrm{SP}_0]\!] \supseteq \mathit{Mod}[\![\mathrm{SP}'_1]\!] \supseteq \cdots \supseteq \mathit{Mod}[\![\mathrm{SP}'_n]\!]$$

Let $\mathcal{M}_i = \mathit{Mod}[\![\mathrm{SP}_i]\!], \mathcal{M}'_j = \mathit{Mod}[\![\mathrm{SP}'_j]\!]$. If there exists an abstract data conversion $\Gamma(\mathcal{M})$ on $\mathcal{M} = \mathit{Mod}[\![\mathrm{SP}_0]\!]$, then there also exist data conversions between every pair of algebras $A \in \mathcal{M}_i, B \in \mathcal{M}'_j$. These data conversions constitute the abstract data conversion $\Gamma(\mathcal{M}_i \cup \mathcal{M}'_j)$, which is a subset of $\Gamma(\mathcal{M})$.

Proposition 8 : abstract data conversions may be restricted
For all i, j, k, l such that $0 \leq i \leq k \leq n,\ 0 \leq j \leq l \leq m$

$$\Gamma(\mathcal{M}_k \cup \mathcal{M}'_l) \subseteq \Gamma(\mathcal{M}_i \cup \mathcal{M}'_j)$$

is an abstract data conversion.

Proof: This follows immediately from the definition of implementation, which implies that

$$\mathcal{M}_k \subseteq \mathcal{M}_i \text{ and } \mathcal{M}'_l \subseteq \mathcal{M}'_j$$

By restricting $\Gamma(\mathcal{M}_i, \mathcal{M}'_j)$ the way described above, we again obtain an abstract data conversion.

□

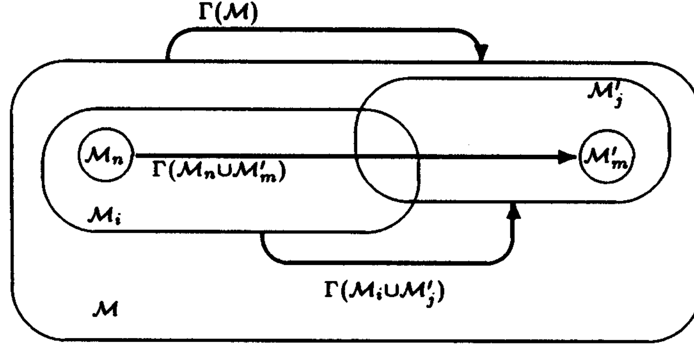

Figure 9: Restriction of an abstract data conversion

The final implementations SP_n and SP'_m specify model classes in which all algebras are equal up to an isomorphism defined by the underlying 'machine' on which the 'code' of these algebras is 'executed'. Consequently, the abstract data conversion $\Gamma(\mathcal{M}_n \cup \mathcal{M}'_m)$ contains only two elements, apart from the identity conversions, namely the data conversions between $\mathcal{M}_n$ to $\mathcal{M}'_m$. This is illustrated in figure 9.

Until now we assumed that a specification SP was implemented by another specification SP′ with the same signature. In practical implementations however, this is mostly not the case; in the refinement step some operations may be added and others may be hidden. Therefore we introduce specification-building operations, in accordance with [6].

$$\begin{array}{ccc} \mathrm{SP} & \overset{}{\underset{\kappa}{\rightsquigarrow}} & \mathrm{SP}' \\ {\scriptstyle Mod}\downarrow & & {\scriptstyle Mod}\downarrow \\ Mod[\![\mathrm{SP}]\!] \supseteq \tilde{\kappa}(Mod[\![\mathrm{SP}']\!]) & \underset{\tilde{\kappa}}{\longleftarrow} & Mod[\![\mathrm{SP}']\!] \end{array}$$

Figure 10: Specification building operator

Definition 5 : specification-building operator
Let SP and SP′ be specifications. A specification-building operator κ between specifications,

$$\kappa : \mathrm{SP}' \mapsto \mathrm{SP}$$

determines a functor

$$\tilde{\kappa} : Mod[\![\mathrm{SP}']\!] \to Mod[\![\mathrm{SP}]\!]$$

A specification-building operator must satisfy

$$Sig[\![\kappa(\mathrm{SP}')]\!] = Sig[\![\mathrm{SP}]\!] \text{ and}$$
$$Mod[\![\kappa(\mathrm{SP}')]\!] = \{\tilde{\kappa}(A) | A \in Mod[\![\mathrm{SP}']\!]\}$$

We say that a specification SP is implemented by a specification SP′ *via a specification-building operator* κ, written $\mathrm{SP} \underset{\kappa}{\rightsquigarrow} \mathrm{SP}'$, if $\mathrm{SP} \rightsquigarrow \kappa(\mathrm{SP}')$. This is illustrated in figure 10.

In [6] several specification-building operations are given, with their associated functors. In this paper we only consider so called constructors [6], but we do not want to go into the details of specific constructors. For our purposes the above definition is sufficient.

We must now revise proposition 8, because we cannot merely restrict abstract data conversions anymore. The sequence of models of partial implementations now becomes

$$Mod[\![\mathrm{SP}_0]\!] \supseteq \tilde{\kappa}_1(Mod[\![\mathrm{SP}_1]\!]) \supseteq (\tilde{\kappa}_2; \tilde{\kappa}_1)(Mod[\![\mathrm{SP}_2]\!]) \supseteq \cdots \supseteq (\tilde{\kappa}_n; \ldots; \tilde{\kappa}_1)(Mod[\![\mathrm{SP}_n]\!])$$

Let $\mathcal{M}_i = (\tilde{\kappa}_i; \ldots; \tilde{\kappa}_1)(Mod[\![\mathrm{SP}_i]\!])$ (see figure 11). Once an appropriate abstract data conversion $\Gamma(\mathcal{M}_n \cup \mathcal{M}'_m)$ between two implementations has been selected, this must be transformed according to the functors corresponding to the specification-building operators $\kappa_1, \ldots, \kappa_n$ and $\kappa'_1, \ldots, \kappa'_m$ used to construct SP_0 from SP_n and SP'_m, respectively. If $Mod[\![\mathrm{SP}_n]\!] = \{A\}$ and $Mod[\![\mathrm{SP}'_m]\!] = \{B\}$, this results in a data conversion

$$\Gamma_{\mathrm{A,B}} : A \to B$$

Proposition 9 : construction of data conversions
The data conversion $\Gamma_{\mathrm{A,B}}$ as described above can be obtained from the abstract data conversion $\Gamma(\mathcal{M})$ and the functors

$$(\tilde{\kappa}_n; \ldots; \tilde{\kappa}_1) \text{ and } (\tilde{\kappa}'_m; \ldots; \tilde{\kappa}'_1)$$

if $\tilde{\kappa}'_1, \ldots, \tilde{\kappa}'_m$ have inverse functors $(\tilde{\kappa}'_1)^{-1}, \ldots, (\tilde{\kappa}'_m)^{-1}$. This is done by the following construction:

$$\Gamma_{\mathrm{A,B}} = (\tilde{\kappa}_n; \ldots; \tilde{\kappa}_1); \Gamma(\mathcal{M}_n \cup \mathcal{M}'_m; ((\tilde{\kappa}'_1)^{-1}; \ldots; (\tilde{\kappa}'_m)^{-1})$$

We omit the proof here; a more thorough treatment of constructor implementations, as well as the technical details of the construction of abstract data conversions, and the existence of inverse functors associated with specification building operators is topic of current research.

[4] José Meseguer and Joseph A. Goguen, *Initiality, induction and computability,* Algebraic methods in semantics (M. Nivat and J. Reynolds, eds.), pp.459-541, Cambridge University Press, 1983

[5] Donald Sannella and Andrzej Tarlecki, *Building specifications in an arbitrary institution,* Proc. Intl. Symp. on Semantics of Data Types, Springer LNCS 173 (1984), pp.337–356

[6] Donald Sannella and Andrzej Tarlecki, *Toward Formal Development of Programs from Algebraic Specifications: Implementations Revisited,* Report ECS-LFCS-86-17, University of Edinburgh, 1986

[7] Nico Verwer, *Data conversions in abstract data types,* submitted to the SION conference 'Computing Science in the Netherlands 88'