Algebraic translations, correctness and algebraic compiler construction

Theo M.V. Janssen,
Computer Science, University of Amsterdam,
Plantage Muidergracht 24,1018TV Amsterdam, The Netherlands
email: theo@fwi.uva.nl

Abstract

Algebraic translation methods are argued for in many fields of science. Several examples will be considered: from the field of compiler construction, database updates, concurrent programming languages, logic, natural language translation, and natural language semantics. Special attention will be given to the notion 'correctness of translation'. In all fields this notion can be defined as the commutativity of some diagram which connects languages, translation and meanings. For algebraically defined compilers, five different definitions are found in the literature. We argue which of these should be considered the 'right' one (it is not the standard choice). We conclude with a first step towards a general algebraic theory of translation.

keywords Translation, correctness, compiler, embedding, view update, parsing, natural language, commutative diagram.

1 Introduction

Translations occur in many fields of science, and between several kinds of languages. One finds them in computer science when a computer program is compiled in a machine code, when a view update is translated in a data base update, but also when a natural language is translated in another one by the computer. It arises when some logic is embedded in another logic, or when meanings are given for the expressions of natural language by defining a translation into logic. In many of these fields the same idea arose: use an algebraic approach. The aim of this article (preliminary version: Janssen (1998)) is to compare the algebraic methods from these different fields and discuss some of the fundamental issues.

Certainly not all translations are intuitively correct, and therefore in many disciplines formal correctness notions are given. Often this notion is based upon the commutativity of some diagram, and if that was not the case, it can be brought in that form. Surprisingly, in the field of algebraic compiler construction there is no consensus on the notion 'correctness': at least five different versions occur in the literature! A large part of this article is about this issue, and arguments in favor of

one correctness notion will be given. It is not the correctness notion that is used in most articles on compilers, but it is the oldest one. This notion is also found in several other fields.

The publications on translation in the different fields discuss the same issues and use related notions. So, there seems to be a common basis for a general algebraic theory of translation. In the this paper a first, small step will be made.

2 Translating from programming language to programming language

A compiler can be conceived of as a translation from a source language SL (for instance a high level programming language) to a target language TL (for instance some assembler language). It has been proposed by several authors to deal with compiler design in an algebraic way. In diagram 1 the components are mentioned that will arise in the discussion: all corners are algebras, and all arrows are homomorphisms. Below we consider one proposal for compiler correctness, in the next sections four other proposals will be considered. I would like to emphazise that the investigations are about this aspect only, and that the conclusions do not diminish the other merits these articles have for the field of compiler construction.

The intuitive ideal about a translation is that it formulates precisely the same information in another language. No information is added, nothing gets lost: the meaning of the target language is, if not identical, at least isomorphic with the meaning of the source language. This ideal is formulated in Polak (1981) who requires *Enc* (encode) to be the identity, and in Mosses (1980*a*) who assumes *Enc* and *Dec* (decode) to be isomorphisms. Correctness is then defined as commutativity of diagram 1.

As we shall see in the next sections, when compiler correctness has to be proven by describing *Enc* or *Dec*, this is never done by proving one of them to be an isomorphism. The explanation is that the involved languages are very different and have different meanings. In the next sections examples will be given which illustrate this point. Therefore it is not surprising that in the final version of Mosses (1980*a*), that is Mosses (1980*b*), a different correctness notion is used (viz. the one from section 3). The ideal of identity or isomorphism is reached only if the situation is designed with that aim (see section 10), or if one takes an abstract point of view (see section 13). As far as I know, these cases do not arise in articles about compilers.

3 The correctness notion of Thatcher et al.

Certainly the most influential proposal for algebraic compiler construction is the one of Thatcher, Wagner & Wright (1979). It defines compiler correctness as commutativity of diagram 2. The proposal of Thatcher et al. is based upon the work of Morris (1973) and aims at correcting, refining and completing that proposal. They do not present Morris' original version (diagram 3); instead they say that his advise was to use diagram 2. This is justified in a footnote where they say that 'Morris' diagram

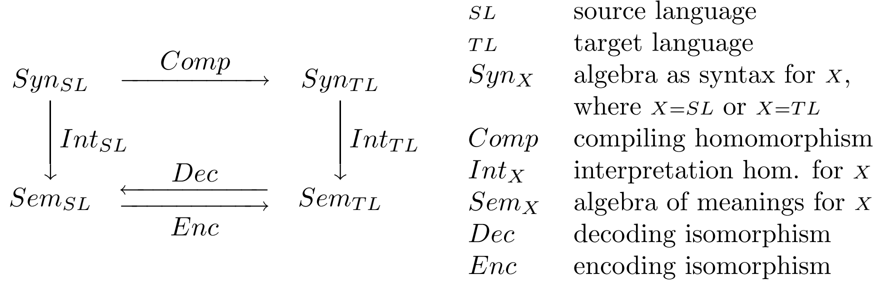


Diagram 1: Compiler correctness according to Polak (1981, p. 17) and Mosses (1980*a*, p. 189): there are isomorphisms *Dec* and *Enc* such that the diagram commutes in both directions.

had $Dec\colon Sem_{TL} \to Sem_{SL}$, though in the text he uses $Enc\colon Sem_{SL} \to Sem_{TL}$'. Thus they suggest that by accident the wrong diagram was incorporated in Morris' article. We shall return to this point in section 4.

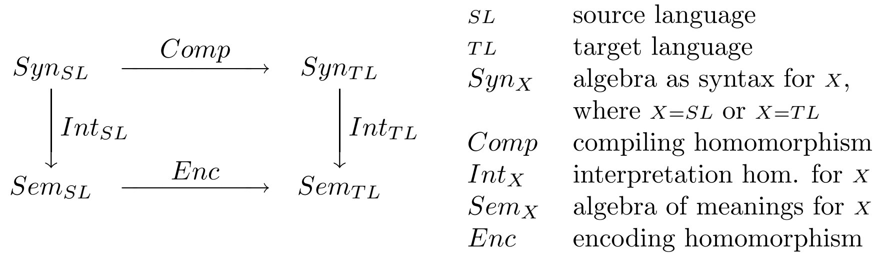


Diagram 2: Compiler correctness according to Thatcher et al. (1979): there is a homomorphism *Enc* such that the diagram is commutative

Let us now consider the example of a compiler given by Thatcher et al. The source language is a fragment of a programming language with 18 syntactic operations (forming for instance assignments, conditionals and the while-construction). Sem_{SL} is a kind of denotational semantics. Its primitive operations are *assign* and *fetch*, together with general algebraic and arithmetical operations. The meanings of the syntactic operations are described by polynomials over the primitive operations. The target language consists of flow charts, and its meanings in Sem_{TL} are unfolded flow charts. As they say, the radical improvement in comparison with Morris lies in this part: making flowcharts algebraic. *Enc* is defined as a mapping from the carriers of Sem_{SL} into corresponding carriers in Sem_{TL}. For instance, the functions from *Environments* to *Environments* are mapped to the functions from $\langle Stacks \times Environments\rangle$ to $\langle Stacks \times Environments\rangle$ that leave the stack unchanged. Next it is proven that *Enc* is a homomorphism. The proof requires

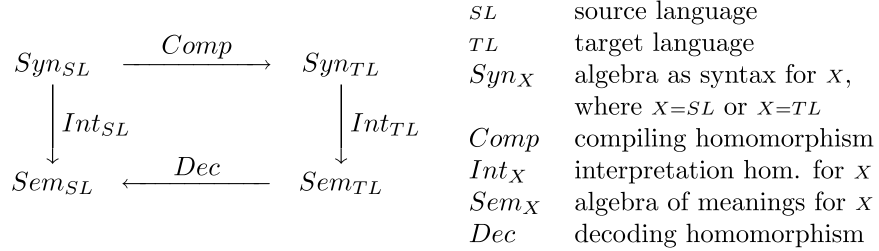


Diagram 3: Compiler correctness according to Morris (1973): there is a homomorphism *Dec* such that the diagram is commutative

the checking of the 18 syntactic operations, and uses many properties of Sem_{SL} and Sem_{TL}. As a consequence both $Enc \circ Int_{SL}$ and $Int_{TL} \circ Comp$ are homomorphisms from Syn_{SL} to Sem_{TL}. Since Syn_{SL} is an initial algebra, there is a unique homomorphism from Syn_{SL} to Sem_{TL}, hence $Enc \circ Int_{SL} = Int_{TL} \circ Comp$, so the diagram commutes.

The definition of compiler correctness as commutativity of diagram 2 is, in my opinion, not satisfactory. In the left hand side of the diagram some programming language is given, together with the intended meaning of this language. The right hand side should tell a machine how to perform the actions described by the programming language. Since a compiled program should do what it has to do according to the semantics of the programming language, going through a compiler should be a way to obtain the originally intended semantics. Hence the meanings of the target algebra should be interpreted in the original semantic algebra of the programming language in order to see whether the compiler yields the intended results. So for a correct compilation there has to be a decoding mapping $Dec\colon Sem_{TL} \to Sem_{SL}$ such that diagram 3 commutes, i.e. the diagram of Morris gives the appropriate definition.

This argument can be illustrated by an example.

Example 1. Let SL be a programming language that has both positive and negative numbers, and has multiplication as operation. Suppose that in the interpretation of TL all information concerning signs is thrown away: Sem_{TL} operates only with positive numbers. Of course, this is not what intuitively would be called a correct compiler. According to the definition of Thatcher et al., this would be a correct compiler since there is a homomorphism *Enc* such that diagram 2 commutes (let the image of a number be its absolute value). Diagram 3 cannot be made commutative: there exists no decoding *Dec* that could achieve this, because a positive number in Sem_{TL} then should have two images in Sem_{SL}. So, according to the definition of Morris, the proposed compiler is incorrect, and this is in accordance with the intuition.
□

This example illustrates that a compiler should not throw away information that

is essential for the source language. What is essential, is of course formalized by the semantic interpretation of the source language.

As a matter of fact, Thatcher et al. admit that their definition is not fully adequate. They say ‘As Barry Rosen has pointed out to us, commuting of diagram 2 is not, in itself, “compiler correctness”. Syn_{TL} and Sem_{TL} could be one-point algebra’s and $Comp, Int_{TL}$ and Enc the unique homomorphisms to those one-point algebras resulting in a commutative square. One possibility around this degenerate case, suggested by Rosen, would be to require the encoding (Enc) to be injective (it is in our case) and that condition is certainly sufficient. We are just not sure at this time that it is necessary.’

Above we argued that it is necessary to require that there is a decoding Dec. If Enc is injective (as Rosen suggested), then it indeed has an inverse Dec, and then Sem_{SL} and Sem_{TL} are isomorphic. Should that be a general requirement for correctness? The examples below illustrate that this certainly is not the case.

Example 2. Suppose SL has the syntactically distinct expressions -0 and $+0$, and both have semantic interpretation: the number zero. Suppose moreover that they correspond with two distinct expressions in TL (again -0 and $+0$), and that their interpretation in Sem_{TL} differs as well (say a different sign bit in their representation in the memory). Let the decoding homomorphism Dec map them to the same value Sem_{SL}: the number zero.

This compiler would intuitively be considered as correct. Indeed, diagram 3 commutes, and the compiler is correct according the definition of Morris. There is is no encoding homomorphism Enc that makes diagram 2 commutative, so according to the definition of Thatcher the compiler would be incorrect. Note moreover that there is there is no isomorphism between Sem_{SL} and Sem_{TL}.
□

Example 3. Compilers resembling the one in example 2 were made in the seventies, an example is the CDC cyber. It had two representations for the number zero (positive and negative zero). Negation of a number (a string of bits) was very simple: replace each 1 by a 0, and each 0 by a 1. This number representation system was called ‘one’s complement’. The main disadvantage was that arithmetical operations yielded $+0$ or -0 depending on the operands. (analysis by Tanenbaum (1975), reported by van der Meer (1994)). Later computers (e.g. the IBM 360) used another system (‘two’s complement’) which has only one representation for zero. The disadvantage is that the system is not symmetrical: the number of positive numbers is not equal to the number of negative numbers (for a discussion of the two systems, see Tanenbaum (1976)). It is clear that for the compilers with one’s complement arithmetic, there is no encoding homomorphism Enc that makes diagram 2 commutative.
□

This discussion shows that requiring Enc to be injective is not the right solution. The notion ‘correct compiler’ is not formalized by diagram 2, but by diagram 3, which requires a decoding homomorphism.

4 The correctness notion of Morris

In section 3 it appeared that the definition of Morris was the correct one. But what about the suggestion of Thatcher et. al. that the diagram in Morris' article was not the intended one? Indeed, all the technical work in Morris' article is about the encoding function *Enc* from source semantics to target semantics. However, he gives explicitly his argument: 'It proves more convenient to define an "encoding" function $Enc: Sem_{SL} \to Sem_{TL}$ than one in the opposite direction; it will be necessary to prove as a final step in proving the correctness to show that *Enc* has a decoding inverse $Dec: Sem_{TL} \to Sem_{SL}$ $[\cdots]$' (Morris 1973, p. 150, +15). Such a statement is repeated after the definition of *Enc* [p. 150, -2]. These quotations show that the occurrence of the decoding *Dec* in the diagram was on purpose, and not some printing error. As a matter of fact, Morris could prove properties of *Enc* instead of properties *Dec* because a situation like the one from example 2 does not arise in his fragment.

Morris' correctness diagram can also be found in earlier publications (Burstall & Landin (1969) and Milner & Weyrauch (1972)) and in Chirica (1976). The great influence of Thatcher et al. (1979) appears from the fact that their approach is followed without discussion in almost all later publications. Significant in this respect is the change from the definition in Mosses (1980*a*) to Thatcher's definition in Mosses (1980*b*). The publications which I found with a correctness diagram, have almost all the same diagram as Thatcher. They are Polak (1981), Dybjer (1985), Royer (1986), Tofte (1990), and Meijer (1992). The exception is T. Rus, who has his own proposal (see section 5).

One might wonder why the original position was so easily abandoned. An explanation could be the influence of category theory. By category theory one is challenged to construct pullbacks, and the encoding homomorphism turns the diagram into such one. In any case, some authors tried (in personal communication) to explain Thatcher's compiler definition with this category-theoretic argument. An additional factor is probably, that the examples discussed in the articles have an injective encoding (this is not made explicit in the articles, though). In general, however, the encoding is no function at all, witness examples 2 and 3.

5 The correctness notion of Rus

5.1 The framework

The compiler definition of Rus can be found in several of his publications (Rus 1980, Rus 1987, Rus 1991, Rus & Halverson 1994). These articles are in two respects of a different nature than the previously discussed ones.

Firstly, his aim is not a definition of compiler correctness, but, as the title of Rus (1991) expresses 'the algebraic construction of compilers'. In his perspective (pers. comm.), one should not make somehow a compiler and prove its correctness afterwards. Therefore his definition of compiler includes correctness; there is no separate correctness definition. In order to be consistent with the previous discussion, we shall separate the two notions and consider the compiler as a translation,

where the commutativity of a diagram expresses its correctness. Since Rus' aim is to construct compilers in an algebraic way, there is a lot attention for (algebraic) tools, such as tools for parsing. Especially Rus & Halverson (1994) and Rus (1995) are devoted such issues.

Secondly, Rus has a fundamentally different perspective on the relation between syntax and semantics. In his view, meaning is primary: one knows what one intends to say, and then one finds a syntactic expression for formulating it. This view is introduced in Rus (1980), and worked out in Rus (1991). It is the motivation for the introduction of a partial learning function $Lrn\colon Sem \rightarrow Syn$. That is a user-dependent function that is defined for the semantic objects the user has learned to express in the language. The partiality may formalize the situation that learning is not completed, but also that for some elements of the semantic domain there is no syntactic expression available.

This view causes a difference in the kind of diagram that defines compiler correctness, see diagram 4. The left hand side of the diagram is intended to express the consistency of the communicator's interaction with his universe of discourse, and the right hand side the consistency of the interaction among communicators (more explanation is not provided). Correctness requires that there is a pair consisting of homomorphisms *Enc* and *Comp* such the diagram commutes.

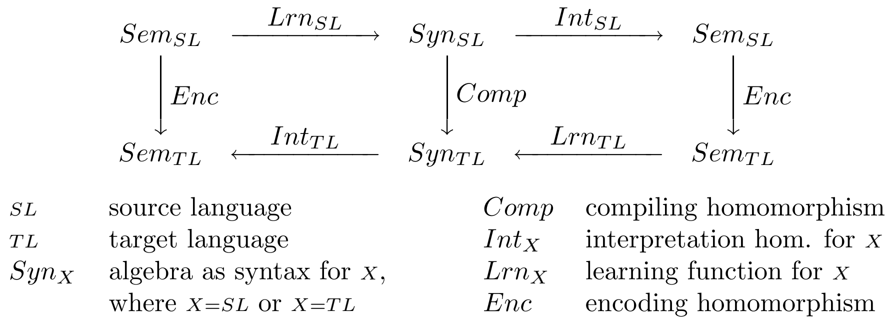


Diagram 4: Compiler correctness according to Rus (1991): there has to be a pair $\langle Enc, Comp\rangle$ such that the diagram commutes

5.2 Discussion

Several questions arise concerning the learning function *Lrn*.

* The partial learning function suggests that expressions (in their relation with concepts) can be learned one by one, and that the learned expressions form some arbitrary subset of the language. Such a way of learning cannot explain how a language user can express programs she has created just before and has never seen expressed. In philosophy of language a related issue arises. A classical quotation is the following; it is from 'Compound thoughts' (Frege 1923), in the translation of Geach & Stoothoff:

> 'It is astonishing what language can do. With a few syllables it can express an incalculable number of thoughts, so that even a thought grasped by a terrestrial being for the very first time can be put into a form of words which will be understood by someone to whom the thought is entirely new. This would be impossible, were we not able to distinguish parts in the thoughts corresponding to the parts of a sentence, so that the structure of the sentence serves as the image of the structure of the thoughts.'

Formulated for the present situation, it means that one learns what the basic concepts of the programming language are (meaning and form), and how these can be combined to larger expressions. So one learns a semantic and a syntactic algebra. Basic parts in the semantics correspond with basic parts in the syntax, and the same for ways of combine parts. So *Lrn* is not an arbitrary function from semantics to syntax, but a homomorphism. The aspect of partiality can be accounted for by the assumption that one learns a subalgebra of all possible meanings and a subalgebra of the expressions. As a matter of fact, in an earlier paper (Rus 1987) *Lrn* is defined as a homomorphism, but this is not repeated later.

∗ The direction of the learning function is remarkable: from meanings to expressions; usually meaning are assigned to expressions. The motivation (that meanings are primary) is appealing, and may be in a some sense correct. But this is, at least in the present context, not properly formalized by the introduction of *Lrn*. If there is such a function it means that no two expressions have the same meaning (if they are in the range of *Lrn*). However, such a situation is not uncommon (e.g. 7 and 007 have the same meaning). So here the framework is too restrictive.

  As a matter of fact, in the technical analysis given below, it will turn out that the situation that the language has synonyms is not excluded. Then a switch is made to a quotient algebra of the syntax. But that is a remarkable theory of learning, because it formalizes that a set of synonyms is learned in one step.

∗ It will be difficult to present such a function *Lrn* without relying on a given language. Suppose the semantic domain is introduced with a general construction, e.g. all functions from numbers to numbers. How could one specify the ones which have been learned? It cannot be all functions, because their cardinality is uncountable, whereas languages usually have denumerable expressions. The functions themselves have no structure and thus do not give information on how to express them. Hence it seems that some language is needed to select the relevant functions in the semantic domain. But then we are back in the situation that language is the primary object.

  For these reasons, some examples of learning functions would be helpful. But Rus does not present such examples. The reason is: 'Since only the syntactic representations of the communication messages between communicators speaking languages SL and TL are actually handled during their communication, only the syntax map *Comp* is actually performed by the compiler' (Rus

1991, p. 298). Also in other contexts, neither in computer science, nor in natural language theory, I have seen mappings from *Sem* to *Syn* which are defined without first giving a language with a meaning assignment.

For these reasons, I disagree with the framework of Rus, and reject his definition of compiler correctness.

5.3 Technical aspects

Below, we shall investigate some technical aspects of Rus' proposal. It will turn out that in fact his correctness notion is a variant of the one proposed by Thatcher, and that related counterexamples can be given. So not only the fundamental objections from section 5.2, but also technical points argue against Rus' correctness notion.

Rus' definition of a programming language is

Definition 4. A programming language is a triple:
$PL = \langle Sem,\ Syn,\ Lrn\colon Sem \rightarrow Syn\rangle$, where

* Sem is a universal algebra, the semantics
* Syn is a term algebra of the same signature as Sem
* Lrn is a partial mapping, called learning function, such that there is a homomorphism $Int\colon Syn \rightarrow Sem$ such that for all m in Sem for which Lrn is defined, holds that $Int(Lrn(m)) = m$.

□

Besides this definition, more is said about the relation between Lrn and Int. Suppose that two elements a_1 and a_2 of Syn have the same meaning: $Int(a_1) = Int(a_2)$. Then a quotient algebra of the syntax is used (induced by the congruence relation 'have the same values under Int'). So, without loss of generality, it can be assumed that two distinct elements have distinct meanings (Rus 1991, pp. 281 - 282). Hence the homomorphism Int from Syn to Sem is a monomorphism, and the image of Int is a subalgebra of Sem which is isomorphic with Syn. The learning function Lrn is a partially defined inverse of Int, hence a partially defined homomorphism.

With this information we can reconsider the compiler definition of diagram 4. For the commutativity requirement only the domain and range of Lrn play a role (both for SL and TL). Let Sem' be the domain of Lrn, and Syn' its range. Let Int' be the restriction of Int to Syn'; so Int' is the inverse of Lrn. Note that Sem' and Syn' are sets, which, due to the partiality of Lrn, need not be algebras. The requirement of commutativity of the diagram 4 means that its two subdiagrams have to commute. Hence the compiler definition of diagram 4 can be represented as in diagram 5.

Since Int'_{SL} is the inverse of Lrn_{SL} the left hand side of diagram 5 commutes only if the left hand side of 6 commutes, and the same for the commutativity of their right hand-sides. One recognizes the left hand side of diagram 6 as diagram 2, but clockwise rotated; and the right hand side of diagram 6 as its mirror image.

$$
\begin{array}{ccc}
Sem'_{SL} & \xrightarrow{Lrn_{SL}} & Syn'_{SL} \\
\Big\downarrow Enc & & \Big\downarrow Comp \\
Sem'_{TL} & \xleftarrow{Int'_{TL}} & Syn'_{TL}
\end{array}
\qquad
\begin{array}{ccc}
Syn'_{SL} & \xrightarrow{Int'_{SL}} & Sem'_{SL} \\
\Big\downarrow Comp & & \Big\downarrow Enc \\
Syn'_{TL} & \xleftarrow{Lrn_{TL}} & Sem'_{TL}
\end{array}
$$

Diagram 5: Reformulation of diagram 4: the two diagrams have to be commutative. Due to the partiality of Lrn, the corners are sets, and not necessarily algebras. $Sem'_X = dom(Lrn_X)$, $Syn'_X = range(Lrn_X)$, $Int'_X = Int\lceil range(Lrn_X)$

This shows that the correctness notion of Rus is essentially the correctness notion of Thatcher et al. (1979). The differences are that in Rus (1991) Int_X is injective, and that the corners of the commutative diagram need not be algebras.

$$
\begin{array}{ccc}
Sem'_{SL} & \xleftarrow{Int'_{SL}} & Syn'_{SL} \\
\Big\downarrow Enc & & \Big\downarrow Comp \\
Sem'_{TL} & \xleftarrow{Int'_{TL}} & Syn'_{TL}
\end{array}
\qquad
\begin{array}{ccc}
Syn'_{SL} & \xrightarrow{Int'_{SL}} & Sem'_{SL} \\
\Big\downarrow Comp & & \Big\downarrow Enc \\
Syn'_{TL} & \xrightarrow{Int'_{TL}} & Sem'_{TL}
\end{array}
$$

Diagram 6: The two (identical!) commutative diagrams obtained from diagram 5, using $Int'_X = {Lrn_X}^{-1}$

Since the correctness notion of Rus turns out to be essentially the correctness notion of Thatcher et al., the same objection applies. The commutativity may be trivial because the target language has a trivial semantics (because it is a one-element algebra), or less dramatic, because important information is lost. An example is given below; it is a variant of example 2. That could not be used unchanged, because Int in that example is not injective. As a matter of fact, this counterexample could also be given for the original diagram 4, so without the investigations concerning Lrn and Int. But then the comparison with the other proposals could not have been made.

Example 5. Suppose Syn_{SL} and Sem_{SL} have positive and negative numbers, and suppose that by the compilation $Comp$ in the signs disappear. So both in Syn_{TL} and in Sem_{TL} the numbers have no sign. Such a compiler is intuitively incorrect because it identifies numbers which, according to the semantics of TL, should be different. But diagram 4 and the two diagrams in 5 and 6 are commutative. So according to Rus' definition, the compiler would be correct.
□

6 The correctness notion of Chirica

Chirica (1976, chapter 7) argues for a variant of the definition of Morris, viz. one in which the decoding is a weak homomorphism, (a relation) instead of a homomorphism see diagram 7. The relevant definition is:

Definition 6. $R: X \to Y$ is a *weak homomorphism* if R is a subalgebra of $X \times Y$

□

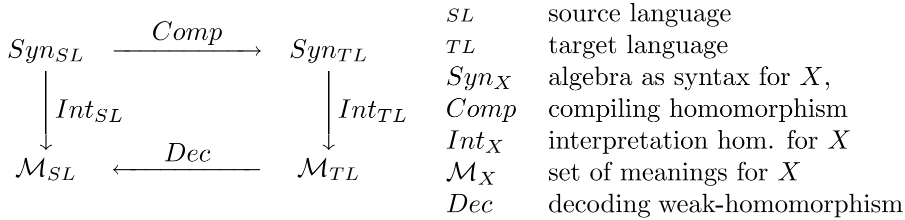


Diagram 7: Compiler correctness according to Chirica (1976): the diagram has to be commutative

In Chirica's approach, the interpretations of programs in the source language are input-output functions. The target language semantics are state transformers. A state is defined as $s = \langle i, m, o, l \rangle$. It gives the current values of the input/output files i and o, the current contents of the memory m, and the current index of the top of the stack.

Chirica provides two arguments for the introduction of weak homomorphisms.

The first argument has to do with 'side effects'. Intermediate results of calculations may be left in the memory, above the current top. For all those (irrelevant) differences in state, separate homomorphisms have to be defined. Chirica argues that it is easier to use weak homomorphisms. This argument will not be considered because it is for convenience only.

The second argument is of a more principled nature. The source language meaning has to be obtained from the state transformers by a homomorphism. An option is to assume that the execution of a program starts in a state where all memory locations contain the value $undefined$, the output file is empty, and the stack index is 0. Then the homomorphism from target language meanings to source language meanings can be defined by $\mathcal{D}(f) = \lambda i.outputfile(f(i, undefined, \langle\rangle, 0))$ However, then only a *limited* proof of the correctness is obtained: no correctness is guaranteed if the machine starts in a different state. For correctness, a family of homomorphisms is to be considered, viz. a homomorphism for each combination of a memory contents and stack index. And Chirica argues that it is easier to give up functionality, and use a relation approach.

The fundamental issue raised here is what the meaning of a program is. Is it the interpretation in one model or in a class of models? This is a fundamental change of view, and not just a matter of convenience. This becomes evident if one

considers translations between logics, where the interpretation in a class of models is the notion one really is interested in. If interpretation is with respect to one model, then *Dec* is a homomorphism, and if it is with respect to class of models, it is a family of homomorphisms. Now a family of such homomorphisms can be seen as a single homomorphism defined (elementwise) on a set of states, and 'weak homomorphism' is in fact such a homomorphism. So the correctness notion of Chirica is the same as the one by Morris, but for a fundamentally different meaning concept.

7 Translating from view language to database language

In this section we consider a translation problem that arises in connection with databases. Usually individual users of a data base are not allowed to see all the data, let it be the full structure of the data base. They have only access to their own view, which gives a restricted, and maybe modified, perspective. The view facility allows each user to see the database in its own way. The relation with the original data base is given in the view definition, which maps a state of the database into a view state. Instructions which the user performs on his view have to be translated into instructions of the database itself. For queries this goes without complications. For updates this raises problems because there can be several data base updates that correspond to a given view update. Furthermore, the update has to be done in such a way that also after further updates the data base remains in correspondence with the view.

Bancilhon & Spyratos (1981) study the problems mentioned above, and investigate which translations are allowed. Their first step is to formulate the requirements: updates can be undone, the composition of two updates is an update again, and the translation is a homomorphism. These properties are not formulated with commutative diagrams, but their proposal (their section 3) becomes more transparent if we do so.

Definition 7. Let U be an algebra of view updates, where U is closed under the operations composition (;) and right-inverse ($^{-1}$). Let $\mathbf{1}_{DB}$ be the identity on the data base. A *correct translation* is a homomorphism with the following two properties:

1. $T(u; u^{-1}) = \mathbf{1}_{DB}$
2. Diagram 8 commutes.

□

The solution of Bancilhon and Spyratos consists of a characterization of the situations in which a correct translation is possible. Generally, in such a situation several choices for a translation are possible, and each choice can be seen as an update strategy. Their solution (paraphrased in theorem 13 below) is simplified by a reformulation in a more abstract algebraic terminology.

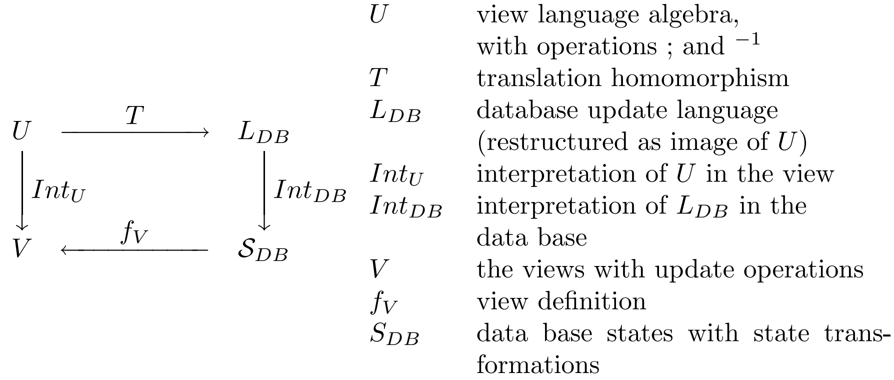


Diagram 8: T is a correct translation of view updates if the diagram commutes (algebraic reformulation of Bancilhon & Spyratos (1981, section 3))

Definition 8. [B.&C. p.559] A view definition consists of

1. a set of relational variables

2. a view defining function f_V from data base states to view states

An immediate consequence is:

Fact 9. Let S be a set of data base states on which f_V defines view V. Then $S/_{f_V}$ is a partition of S.

Definition 10. V_s is the partition element of $S/_{f_V}$ that contains s.

Definition 11. [algebraic paraphrase of B.&C., def 4.4]
W is a *complementary view* of V if for all $v \in S/_{f_V}$ and for all $w \in S/_{f_W}$ holds $|v \cap w| = 1$.

Since each pair (view,complementairy view) determines a unique element of $\mathcal{S}$, we have

Fact 12. Let W be a complementary view of V. Then $\mathcal{S}$ is isomorphic with $S/_{f_V} \times S/_{f_W}$.

Now it can be proven that

Theorem 13. [algebraic paraphrase of the main result: B.&C., th. 7.1]
Let U be an algebra of updates of view V. There exists a correct translation of U in the data base if an only if there exists a complementary view W of V such that for all updates $u \in U$ holds $[T(u)](s) = u(V_s)W_s$.

.

8 Translating from concurrent language to concurrent language

Shapiro (1991) presents a general method to compare languages and his particular aim is to compare concurrent programming languages. Such languages are difficult to compare because they use different notions of communication and synchronization and different models, and therefore their semantic models are often irreconcilable. The method Shapiro presents, is based on algebraic embeddings.

The central notion in his approach is 'observable behavior': which in the present context means 'state transitions'. In another paper (Moscowitz & Shapiro 1993) he applied the method to compare languages defined by machines (Turing machines and finite automata). Here the behavior of concurrent programming languages with respect to their parallel composition operator is the theme. The relation 'have the same observable behavior' defines an equivalence relation on the programming obtained, and this relation can be characterized as the kernel of interpretation function Ob . These behaviors can be compared, and thus give a comparison of the languages. His key definition is given below; he calls 'sound', what we called 'correct', otherwise it is identical to his proposal.

Definition 14. A translation homomorphism Tr is *correct* if there is a Dec such that diagram 9 commutes.

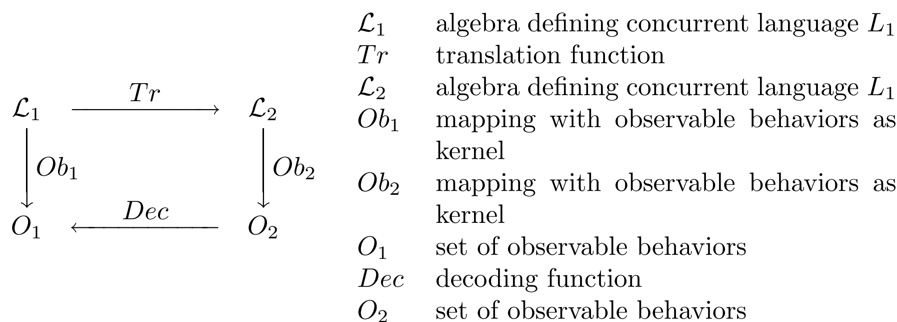


Diagram 9: Correct translation of concurrent languages (Shapiro 1991, p. 200)

A stronger notion is what he calls *faithful*: when Dec has an inverse. The meaning assignment Ob is *compositional* if that mapping defines a congruence relation. For these notions he proves some theorems, e.g. conditions when a correct translation is also faithful.

His main theorems state of two properties of concurrent languages that they are preserved under correct translations: 'interference freedom' and 'connection hiding'. For several programming languages it is proven either that they have such a property, or that they do not. Thus this method of comparison primarily yields many negative results: concurrent languages for which no embedding is possible. Positive results are more difficult to obtain because then details of the concurrent behavior have to be considered. That is done for some languages in Shapiro (1992).

Together with embeddings from the literature, it gives a catalogue in which 22 concurrent languages are compared.

Note the resemblance of Shapiro's work with the translations used in logic (section 9). Both aim at comparing the relative strength of languages, are interested in comparing only one central notion (observable behavior, respectively truth in models), and in the interpretation all other aspects are neglected.

9 Translating from logic to logic

There are many logical languages, and between several of them translations have been defined. The purpose of such translations is to investigate the relation between the logics, for instance their relative strength or their relative consistency. If one considers the method behind such translations, it turns out that (almost) always the algebraic method is used. We shall consider a famous example: Gödel's translation (denoted Gt) of intuitionistic propositional logic into modal logic (e.g. van Dalen (1986), Epstein (1990)).

In intuitionistic logic connectives have a constructive interpretation. For instance $\phi \to \psi$ could be read as 'given a proof for ϕ, it can be transformed into a proof for ψ'. The disjunction $\phi \vee \psi$ is read as 'a proof for ϕ is available or a proof for ψ is available'. Since it may be the case that neither a proof for ϕ nor for $\neg\phi$ is available, it is explained why $\phi \vee \neg\phi$ is not a tautology in intuitionistic logic . This explanation has a modal flavor, made explicit in the translation Gt into modal logic S4. In the clauses of the translation the negation does not occur because in intuitionistic logic $\neg\phi$ is defined as an abbreviation for $\phi \to \bot$, where $\bot$ is 'absurdum'.

1. Gt(p) $= \Box p$,   for p an atom
2. Gt($\phi \vee \psi$) $= Gt(\phi) \vee Gt(\psi)$
3. Gt($\phi \wedge \psi$) $= Gt(\phi) \wedge Gt(\psi)$
4. Gt($\bot$) $= \bot$
5. Gt($\phi \to \psi$) $= \Box[Gt(\phi) \to Gt(\psi)]$

Note that the translation of $p \vee \neg p$ is $\Box p \vee \Box\neg\Box p$ (which is not a tautology in modal logic).

It is straightforward to formulate this translation in an explicit algebraic format. Then the intensional logic operator $IL_\vee$ (which puts $\vee$ between its two input arguments) corresponds with $ML_\vee$. And $IL_\to$ corresponds with the polynomial operator $ML_\Box(ML_\to(X,Y))$. So the translation is, algebraically spoken, not in IL, but in a from IL polynomially derived algebra.

Logics are usually defined by a proof system, and the traditional notion related with translations is 'interpretation in another logic': a translation interprets a logic if every formula provable in the original logic is provable in the translation. If it also is vice versa, it is called 'faithful'.

Definition 15. A translation Tr from logic L_1 in logic L_2 is called an *interpretation* of L_1 in L_2 if:

$$\vdash_{L_1} \phi \quad \Longrightarrow \quad \vdash_{L_2} Tr(\phi)$$

A translation is called a *faithful interpretation* if

$$\vdash_{L_1} \phi \quad \Longleftrightarrow \quad \vdash_{L_2} Tr(\phi)$$

□

In case the (proof systems of the) logics are sound and complete with respect to a class of models, the interpretations defined before can be described semantically: the translations of valid formulas in the one logic should be valid in the other logic and vice versa (valid = true in all models). This is not the traditional perspective on translation, but it is not difficult to formulate corresponding semantic notions.

Definition 16. A translation Tr from L_1 in L_2 is called a *semantic interpretation* of L_1 in L_2 if:

$$\models_{L_1} \phi \quad \Longrightarrow \quad \models_{L_2} Tr(\phi)$$

A translation Tr from L_1 in L_2 is called a *semantic faithful interpretation* of L_1 in L_2 if:

$$\models_{L_1} \phi \quad \Longleftrightarrow \quad \models_{L_2} Tr(\phi)$$

□

We might try to formulate an algebraic correctness definition resembling the ones used in the other sections. There are however complications, which are mentioned below.

Definition 17. A translation Tr from L_1 to L_2 is *correct* if there is a mapping m such that diagram 10 commutes.

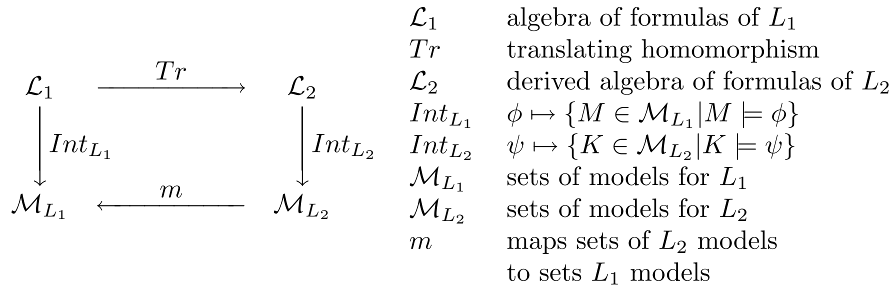


Diagram 10: Tr is correct if the diagram commutes (Suggestion)

There are constructions which transform models of intensional logic into S4 models and vice versa (Epstein 1990)[p. 308]), hence: Gt is a correct translation of intuitionistic logic in modal logic.

Definition (16) is not equivalent with the definition (17). Firstly, (17) requires that m is defined for all sets of models that may arise for a formula, whereas (16) only speaks about the valid formulas. Secondly, (17) leaves the possibility open that the embedding of L_1 in L_2 is done in another way than by means of translating valid formulas to valid formulas. Finally, it is not clear whether the interpretation functions can be homomorphism.

These points require further investigations. The work on institutions probably is relevant. Institutions constitute an abstract framework for the study of the relation between specification languages (or programming languages) and their interpretations, and then a diagram like 10 arises, see e.g. Goguen & Burstall (1992).

The method of translating exemplified by Gt, viz. the algebraic method, is the standard method in the field of logic: the definition of translation follows the clauses of the grammar of the source language logic, and for each clause the translation is given by a (possible compound) expression in the target logic. A large number of translations between logics is collected in Epstein (1990, Chapter 10: 'Translations between Logic'. pp. 289-314). Almost all of them are homomorphisms (there they are called 'grammatical translations'), and the few that are not, are also in other respects deviant [p. 313]. It would be interesting to investigate the semantic (model-theoretic) counterparts of such non homomorphic translations.

10 Translating from natural language to natural language

Many methods have been proposed for translating from one natural language to another. The Rosetta project of the Philips Research laboratories (Eindhoven, the Netherlands) has used one that is in that field very special: an algebraic method. The syntax of the source language is organized as an algebra, the syntax of the target algebra is a similar algebra, and translating is an *iso*morphism. Their approach is illustrated by the following simplified example.

Consider sentence (1), which has (2) as translation in Dutch.

(1) Peter does not sing.

(2) Peter zingt niet.

The sentences have different syntactic structures: in English there is an auxiliary verb (*do*) that has no counterpart in the Dutch sentence. If one would design for each language separately context free rules producing the respective sentences, then the grammars would not be isomorphic. Nevertheless, in Rosetta the sentences are generated by isomorphic algebras.

The generators of an algebra E for this fragment of English are *Peter* and *to sing*. For Dutch the corresponding generators are *Peter* and *zingen*. E has an operator $R_{E,1}$ that produces from the two generators sentence (3), and $R_{D,1}$ produces likewise (4).

(3) Peter sings.

(4) Peter zingt.

Furthermore there is an operator $R_{E,2}$ that takes as input a sentence and yields its negation. This is not a straightforward rule because the rule has to find the finite verb, move it to another position, and insert *does* and *not*. The Dutch rule $R_{D,2}$ is simpler. So we have:

(5) $R_{E,2}(R_{E,1}(\text{Peter, to sing}))$ = Peter does not sing.

(6) $R_{D,2}(R_{D,1}(\text{Peter, zingen}))$ = Peter zingt niet.

The left hand sides of (5) and (6) describe how the sentence is formed. In (5) it says that $R_{E,1}$ is applied to two generators (*Peter, to sing*), and next $R_{E,2}$ is applied to the result. In algebra the left hand side of (5) is called a 'term', so a term represents a derivation of an expression. The terms corresponding with an algebra A form an algebra themselves, called the term algebra, denoted as T_A.

As one sees, the terms (derivations) in (5) and (6) are isomorphic. This is also the case for the (large) fragments described in the Rosetta system. The isomorphism became possible by adopting the following points of view:

* The algebras are designed, and not discovered as (innate) properties of the mind. The latter is the opinion of a prominent tradition in linguistics.

* The design is guided by semantic insights: a syntactic operator corresponds with a meaning operation. This aspect constitutes a difference with many grammatical models in the linguistic tradition or in computational linguistics. In section 11 more information will be given about meanings for natural languages.

* Operators are powerful, they do more than just concatenation. They do not necessarily correspond with context free rules, so they differ from the operators used in computer science (see section 12).

* Algebras for different languages are tuned. The algebra for a language is not necessarily the algebra that would be designed for the language when considered in isolation: sometimes a decision for one language is influenced by phenomena in the other language.

There are some properties of natural languages that cause differences in algebraic respects with the framework defined in other sections.

1. Natural languages are ambiguous.
   Not all expressions of natural language have a unique meaning, and therefore expressions often do not have a unique translation. The algebra for a language is designed in such a way that an expression which has two different meanings, can be formed in different ways. It may be formed from different generators, or using different operators (or both). So, in algebraic terminology, two different terms may represent the same expression of the language. Hence the

algebra for the source language is not an initial algebra. Since differences in the way of formation of an expression may correspond with differences in translation, the translation homomorphism is not based on the algebra for the source language, but on the term algebra that corresponds with that syntactic algebra. This is by its nature an initial algebra. For the target language the same argumentation applies, so the range of the translation is the term algebra which corresponds with the algebra for the target language.

2. Natural languages are not context free.
   Traditionally it is claimed that natural languages are not context free. Most of the arguments are shown to be incorrect in Pullum & Gazdar (1982), and only in some cases the non context freeness holds (see some contributions in Savitch, Bach, Marsh & Safran-Naveh (1987)). But although very large fragments of natural language (when considered as strings) are context free, several grammatical theories use non context free rules in order to express regularities in natural language. And also the algebra for Rosetta has (influenced by semantic considerations) operators which have much more power than context free (see section 12).

3. Natural languages have synonymous expressions.
   One expression may have several equivalent translations. Rosetta aims at obtaining all possible translations and (distinctly from most translation systems) does not select, by some criterion, one of those. This situation does not only arise for words, but also for operators: one construction in the source language can sometimes be translated by several constructions in the target language. Furthermore, different expressions may have the same translation. So there is a many-many correspondence between source and target language. For these reasons, the translation is defined between sets of expressions. In algebraic terminology the situation is as follows. The relation 'are translation equivalent' is in the system a congruence relation on (sub)expressions. This congruence induces a quotient algebra for each of the term algebras, and the translation is defined between these quotient algebras. Due to this quotient construction, the translation homomorphism becomes an isomorphism. That the translation relation is a congruence relation is partially due to translation properties of natural language, but also due to the design of the algebra.

These three points give rise to suggestions that might be useful for algebraic compiler construction:

1. Programs and statements in programs are not ambiguous, but subexpressions of them are. For instance, if x and y are integers, an expression like $x + y$ denotes an integer value, but in case it is argument of an operator which asks for a real number as input it is considered to denote a real number. So the context disambiguates. Such coercions are usually not considered in the examples presented in articles on algebraic compiler construction. In order to apply the algebraic method to ambiguous expressions, it would be useful to use the term algebra as domain for the compilation homomorphism. Already Knuth (1968) mentions the possibility to use term algebras. Chirica (1976)

deals with a realistic fragment of a programming language and explicitly takes the term algebra as domain for the translation.

2. Synonyms arise in programming languages as well. The example of 7 and 007 is already mentioned in section 5, where also the quotient construction was used. Another example is that + and $\underline{plus}$ may denote the same operation. But the situation that two operations have the same meaning (and compilation) is mostly avoided in programming languages.

3. The algebraic approach turned out to be applicable to non-context free languages. That fact is relevant for algebraic compiler construction and will be discussed in section 12.

A translation from one natural language into another should to be correct. And by correctness is of course understood that the original and the translation have the same meaning. Since natural languages have an infinite number of sentences, correctness of translation is a property on an infinite set. The algebraic method reduces this to a finite property: if the translations of generators and of operators are correct the correctness for all sentences follows. Of the Rosetta system it is assumed, based on intuitions about translations, that the generators and operators are translated correctly. From that, the correctness of the whole system follows. This guarantee is something that other translation systems do not have, and is one of the advantages of the algebraic method. In proving compiler correctness the same situation arises, and the same guarantee can be given.

The algebraic structure of Rosetta is described extensively in chapter 19 of Rosetta (1994). It is summarized in diagram (11).

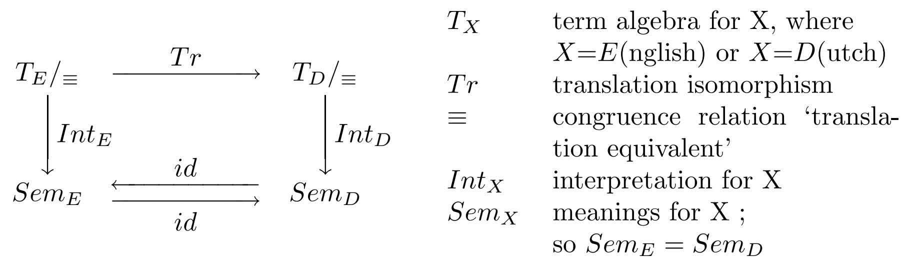


Diagram 11: The algebraic structure of Rosetta for translating from English to Dutch. The translation is correct if the diagram commutes in both directions (cf. Rosetta (1994, ch. 19))

11 Translating from natural language to logic

11.1 Montague grammar

Semantics of natural language is traditionally studied in the field of philosophy of language. Often meanings of natural language expressions are represented in some

logic. For long, say until 1975, in all articles it was more or less stipulated which formula was the correct meaning representation of a given sentence (its 'logical form'). This situation has been characterized as: it seemed that a 'bilingual logician', who knew logic and who knew natural language, had provided the formula. An opinion often heard (the 'misleading form thesis') was that there exists a great difference between the sentence and its logical representation. Therefore it was proposed to design for certain purposes a 'purified natural' language. So natural language and logical languages were two worlds, with only loose connections.

A radical change in this situation was brought by Richard Montague, a mathematical logician. He developed a method to relate natural language and logic in a systematic way. Montague (1973) presented a semantical interesting fragment of English and provided it with a model-theoretic interpretation through a systematically translation into logic. It became, for the first time in history, possible to calculate which meaning is associated with a given sentence, and to make predictions concerning meanings of sentences.

His method was presented in Montague (1970), and it is the same algebraic method followed in the other sections of this paper. The syntax of the natural language is a many sorted algebra, and meaning assignment is a homomorphic translation into a logical language. The domain of this homomorphism is not the syntactic algebra itself, but the corresponding term algebra (the algebra of derivations). That makes it possible to account for ambiguities that arise in natural languages, e.g. the scope ambiguity of *Every man loves a woman*, see section 12. Different readings correspond with different ways of production, so with different terms.

The method of Montague grammar is illustrated by the simplified treatment of sentence (7).

(7) John and Mary walk

The syntactic algebra has three generators: *John* and *Mary* of sort *PN* (Proper Name), and *walk* of sort V (verb). Other sorts are *NP* (Noun Phrase) and S (Sentence). The operators are (for R_1 and R_2, see section 10):

1. $R_1\colon NP \times V \to S$, where $R_1(\alpha, \beta) = \alpha\ \beta$
2. $R_3\colon PN \times PN \to NP$, where $R_3(\alpha, \beta) = \alpha\ \mathit{and}\ \beta$

So the production of (7) is described by the term:

(8) $R_1(R_3(\mathit{John, Mary}), \mathit{walk})$

In Montague's original paper, sentences are translated into intensional logic. That is a higher order modal logic with lambda abstraction. For simplicity, we translate here into extensional predicate logic, enriched with lambda abstraction. The logic has one predicate: $WALK$, and two constants: j and m. The proper names translate into the corresponding constants, and the verb in the corresponding predicate. So the translation Tr of the generators is:

$$Tr(\mathit{John}) = j,\ \ Tr(\mathit{Mary}) = m,\ \ \text{and } Tr(\mathit{walk}) = WALK$$

The operators corresponding with R_1 and R_3 are respectively:

1. T_1: $Bool^{Pred} \times Pred \to Bool$, where $T_1(\gamma, \delta) = \gamma(\delta)$
2. T_3: $Indiv \times Indiv \to Bool^{Pred}$, where $T_3(\alpha, \beta) = \lambda P[P(\alpha) \wedge P(\beta)]$

So the translation of *John and Mary* is:

(9) $\lambda P\,[P(j) \wedge P(m)]$

And of the sentence *John and Mary walk*:

(10) $\lambda P\,[P(j) \wedge P(m)](\mathit{WALK})$

This can be reduced (by lambda conversion) to:

(11) $\mathit{WALK}\,(j) \wedge \mathit{WALK}\,(m)$

A significant point in the above example is the translation of *PN*-conjunction (R_3): it is an operator (T_3) that is defined by means of a polynomial expression. In larger fragments that situation would arise frequently: logic has few operators and constants, whereas natural language needs a lot. So the algebra NL for natural language is not similar with the algebra L for logic. New operators and constants are defined by means of polynomial expressions, and thus within L a reconstruction L' is made of T_{NL}. So T_{NL} is translated onto a polynomially derived algebra L'. Then the interpretation of L determines a unique interpretation of L'. This expressed in:

Theorem 18. (Montague 1970, p. 225) Let L be an algebra (for logic), $\mathcal{I}$ a homomorphism from L to some algebra $\mathcal{M}$, and let L' be an algebra obtained from L by replacing its operations by polynomially defined operations. Then there is a unique algebra $\mathcal{M}'$ such that there is a homomorphism $\mathcal{I}'$ from L' to $\mathcal{M}'$, where $\mathcal{I}'(a) = \mathcal{I}(a)$ whenever $\mathcal{I}'(a)$ is defined.

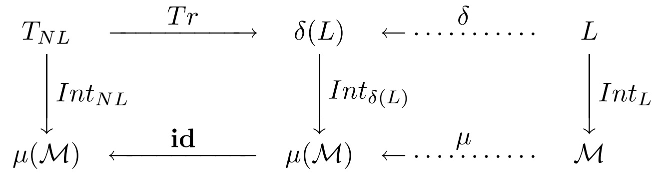
This theorem is the background of the following definition of a Montague grammar. The algebraic structure of a Montague grammar is presented in diagram (12).

Definition 19. A Montague grammar consists of a syntactic algebra NL, a logical algebra L, a polynomial derivor δ and a homomorphism from T_{NL} to $\delta(L)$.

11.2 Correctness

Of course, the meaning assignment is not an arbitrarily chosen one: it has to yield the 'correct' meaning. One might expect that this means that a meaning assignment has to capture our intuitions concerning meanings of phrases. Indeed, this was the case for the meaning assigned to *John and Mary walk*. For certain types of expressions, and for simple sentences, one might base formal meanings directly on intuitions, but in many cases it becomes problematic. The intuition may point in the wrong direction, or there might be no intuition at all, especially for subexpressions of sentences. For instance, the meaning of *only* cannot be something like 'there is precisely one', because it does not only occur in phrases like *only John* but also in *only John and Mary* and *only man*.

The solution is to require meanings to formalize intuitions about entailment relations between sentences. A classical case from Montague (1973) is: sentence (12) entails (14), whereas (13) does not entail (14).



NL	algebra for Natural Language	Int_X	interpretation of X
T_{NL}	terms over the algebra NL	μ	model-theoretic counterpart
Tr	translation homomorphism		of δ
L	algebra of logic	$\mathcal{M}$	model for L
δ	derivor which restructures L	$\mu(\mathcal{M})$	induced model for $\delta(L)$
$\delta(L)$	derived logical algebra	**id**	identity mapping

Diagram 12: The algebraic structure of Montague grammar (cf. Montague (1970) section 5, and Janssen (1986))

(12) John finds a unicorn

(13) John seeks a unicorn

(14) There exists a unicorn

A newer, intricate example is from Groenendijk & Stokhof (1982): from sentences (15) and (16) it follows that (17).

(15) John knows whether Mary comes.

(16) Mary does not come.

(17) John knows that Mary does not come.

This example illustrates again that intuitions concerning meanings of natural language expressions are not always available: what would be the intuition about *whether Mary comes*, or about *that Mary comes*? Based upon intuitions concerning *meaning entailments*, model-theoretic interpretations are defined that can account for the semantic relation between (15), (16) and (17).

The principle behind this heuristics is expressed in a famous quotation from Lewis (1970):

> In order to say what a meaning is, we may first ask what a meaning does, and then find something that does that.

Montague grammar traditionally is a form of possible world semantics. Sentences are interpreted as sets of possible worlds (with the moments of time as parameter), and entailment between two sentences corresponds with set inclusion (for the same parameter value). For instance, for every moment of time, the set of worlds in which (12) is true, forms a subset of the set for which (14) is true. This is not the case for (13) and (14). The examples with *know* require a formalization of the entailment relation that is more complex.

The entailment perspective on correctness is known in Montague grammar, but its formalization, as the commutative diagram (13) is new.

$$\begin{array}{ccc} {T_{NL}}^n & \xrightarrow{\quad Tr \quad} & L^n \\ \Big\downarrow \mathcal{E}nt & & \Big\downarrow Int_L \\ \{ent,\ nent\} & \xleftarrow{\quad Incl \quad} & W^n \end{array}$$

${T_{NL}}^n$	n-tuples of terms for natural language	Int_L	interpretation of L
Tr	translation homomorphism	W^n	n-tuples of meanings: functions from time points to sets of possible worlds
L^n	n-tuples of formulas from logic	$Incl$	model-theoretic formalization of entailment (for $n = 2$ this is $\subseteq$)
$\mathcal{E}nt$	intuitions concerning entailment (*ent*) and non-entailment (*nent*)		

Diagram 13: Tr gives a correct translation of natural language sentences into logic if the diagram commutes.

12 Algebra's and context-free grammars

12.1 Traditional view

Several authors in the field of computer science assume that algebras can only be used for context-free languages; for instance, Chirica (1976, p. 10), Rus (1991, p. 295), Tofte (1990) and Rus (1976)(entitled: 'Context-free algebras [...]'). In one of the first journal articles on the algebraic approach to programming languages, many sorted algebra's and context-free grammars are connected as follows (Goguen, Thatcher, Wagner & Wright 1977):

Definition 20. Let G be a context free grammar with V_N as non terminal and V_T as terminal symbols. Hence the context free rules are of the form: $A \rightarrow w_1 B_1 w_2 B_2 \cdots B_n w_{n+1}$, where $A, B_1, B_2, \cdots B_n \in V_N$ and $w_1, w_2, \cdots w_{n+1} \in V_T^*$. Then G is made into a V_N-sorted algebra by introducing for each rule an operator R of type $\langle B_1 B_2, \cdots B_n, A\rangle$, where $R(\alpha_1, \alpha_2, \cdots \alpha_n) = w_1 \alpha_1 w_2 \alpha_2 \cdots \alpha_n w_{n+1}$.
□

The converse construction gives a context free grammar for a many sorted algebra. Goguen et. al. call context free grammars the most important and general example of their approach. It is remarkable that they allow V_N, V_T and the set of rules to be infinite; usually context free grammars are finite. Rus (1991, p. 295) characterizes the relation between context free grammars and algebras more restrictively by adding finiteness conditions to the above definition.

Programming languages have some context dependent features; a well known example is that every identifier has to be declared before its use. In semantical studies such properties often are assumed to be specified in the syntax. In an algebraic approach this is not desirable, as was pointed out by Chirica (1976). In such a situation meanings are only assigned to full programs, and not to their subprograms because these may contain undeclared identifiers. In this way the meaning assignment is not a homomorphism any more, and the power of structural induction is lost. Chirica considers this issue as a fundamental challenge of the algebraic approach, and therefore aims at solving it. Because the method of inherited and synthesized attributes (Knuth68) is the most successful method to deal semantically with context dependency, Chirica develops a (rather complicated) algebraic version of attribute grammar. Also the fragment of Rus (1995) contains context dependent properties, such as scope. His solution is much simpler than the one by Chirica. Rus deals with them [p. 22] by means of attributes in a special store, hence separated from the algebra.

12.2 Beyond context-free

All points in the above discussion are based upon the assumption that the connection between an algebraic operator and a rule of the grammar is as defined in definition (20). An application of the algebraic approach to natural languages, however, would be very difficult, if not impossible, if that indeed were the case. In algebraic treatments of natural language frequently non context-free operators are used, for instance in the translation system Rosetta, and in Montague grammar. A classical example from Montague (1973) is:

Example 21. Consider the sentence:

(18) Every man loves a woman.

This sentence has two readings:

(19) $\forall x[\mathit{man}(x) \rightarrow \exists y[\mathit{woman}(y) \wedge \mathit{love}(x, y)]]$

(20) $\exists y[\mathit{woman}(y) \wedge \forall x[\mathit{man}(x) \rightarrow \mathit{love}(x, y)]]$

Sentence (18) is the result of two applications of a substitution operator to:

(21) He_1 loves him_2.

The operator substitutes an NP (a Noun Phrase such as *a man* or *every woman*) for a syntactic variable (he_1 or him_2) in an S (Sentence). Differences in the order of substitution determine difference in meaning. The operator reads:

> $S_{14,i}$: $\mathrm{NP} \times \mathrm{S} \rightarrow \mathrm{S}$,
> where $S_{14,i}(\alpha, \beta)$ is obtained by substitution of α for the first occurrence of he_i or of him_i in β, and by substitution of an appropriate pronoun (gender, case) for all other occurrences.

So $\mathrm{S}_{14,2}$(*a woman, He_1 loves him_2*) = *He_1 loves a woman.*
□

Operator $S_{14,i}$ illustrates two features which each cause the power of the algebraic approach to be beyond that of context free grammars:

1. It is an infinite scheme: for each i it defines an operator.
2. The operators perform a substitution somewhere in one of the arguments. So they are operators with more power than context-free rules.

Using the power of replacement, an algebra can simulate a Turing Machine, which proves (for details see Janssen (1997), or Rosetta (1994, chapter 19)):

Theorem 22. For every recursively enumerable language L there is a many sorted algebra such that it generates L.

In the example below an infinite number of sorts and operators is used to take care of the context sensitive property of programming languages that 'all identifiers must be declared before they can be used'.

Example 23. Define an algebra as follows

* Sorts
  - For each $X \subset \{x | x \text{ is an identifier } \}$ there is a sort $\langle P, X \rangle$, viz. of programs with X as set of undeclared identifiers
  - For each identifier x there is a sort $\langle D, x \rangle$ of declarations of identifier x (e.g. **real** y is of sort $\langle D, y \rangle$).
* Operators
  - An infinite collection of operators defined by the following scheme: $R_{Decl\langle X,x \rangle} \colon \langle D, x \rangle \times \langle P, X \rangle \to \langle P, X \setminus x \rangle$, where $R_{Decl\langle X,x \rangle}(\alpha, \beta) = \alpha; \beta$, i.e. the declaration and the program are concatenated with ; as glue.
  - One rule yielding full programs, viz. a rule in which the set of identifiers is empty: $R_{FP} \colon \langle D, \rangle \to FP$, where $R_{FP}(\alpha) = \mathbf{begin}\ \alpha\ \mathbf{end}$.

□

12.3 Parsing

We have seen in the previous sections that in applications to natural language, algebras are used with an infinite number of powerful operators. That raises the question how to find for a given expression the term which describes its generation. The machine translation project Rosetta would not have been possible without an algorithm for this. Finding the terms which describe the generation of an expression will be called 'parsing'.

Since each recursively enumerable language can be generated by an algebra, restrictions have to be imposed, in order to reduce the generative power from recursively enumerable (theorem 22) to recursive languages (or less). The restrictions on the algebras are as follows:

1. **Reversibility condition**
   For each operator R of the algebra A there exists a reverse operator R^{-1}. This reverse operator is not an operator of the algebra, but defined on some superset containing all strings we might wish to parse (including strings that do not belong to A). This operator has the following properties:

   (a) If $\langle x_1, x_2, \ldots, x_n \rangle \in R^{-1}(y)$ then $y = R(x_1, x_2, \ldots, x_n)$. Here $R^1(y)$ consists of n-tuples of elements from the superset, and includes all expressions from A from which y could be formed by one application of R.

   (b) For all y the set $R^{-1}(y)$ is finite.

2. **Measure condition**
   There is a computable function μ hat assigns to each expression its measure; its position in some well founded ordering (usually the natural numbers). Generators are assigned minimal measure. Furthermore:
   If $y = R(x_1, x_2, ...x_n)$, then $\mu(y) > \max(\mu(x_1), \mu(x_2,), \ldots, \mu(x_n))$

3. **Finiteness condition**
   In order to decide whether a given expression is element of the algebra, only a finite number of inverse operators have to be tried in order to find the relevant parses.

A parsing algorithm can be based upon these three properties. Condition 1 makes it possible to find, given the output of an operator, a finite set of possible inputs for the operator. Condition 3 assures that only a finite number of operators have to be tried. Together they define for a given expression a search space of finite size. The process is applied again to each expression in this search space. Condition 2 guarantees termination of this recursive process.

The basic ideas for the parsing algorithm originate from Landsbergen (1981). More information about the algorithm is given in Rosetta (1994): about the implementation in chapters 17 and 18 (by Landsbergen, Leermakers and Rous), and about the algebraic aspects in chapter 19 (by Janssen).

Let us investigate the conditions for the two previous examples:

Example 24. Examples (21) and (23) continued
The three conditions are satisfied:

1. $R_{Decl\langle X,x\rangle}$ satisfies the reversibility condition: the inverse operator has to split its input in a two parts after the last declaration. The operator ${R_{FP}}^{-1}$ has to strip off the added elements. And $S_{14,i}^{-1}$ has to de-substitute.

2. For $R_{Decl\langle X,x\rangle}$ the measure can be the length of the string. For $S_{14,i}$ an abstract measure is required, taking the number of occurrences of the syntactic variable he_i's into account.

3. For parsing a full program, only R_{FP}^{-1} has to be tried. Also for parsing an expression of the sort $\langle P, X\rangle$ only one rule has to be tried, the choice is determined by the first declaration. For $S_{14,i}$ the situation is different, because

a given string can be produced by any out of an infinite set of substitution rules. Here a standardization of the use of indices is required (e.g. only the least unused index has to be tried).

□

For the rules in example (23) the algorithm seems efficient. But there are sorts of operators which may cause a combinatorial explosion: the reverse rule of a simple concatenation operation may yield all possible ways to split a rule in two parts. So further restrictions are needed in order to guarantee efficiency of the parsing algorithm.

The discussion in this section has shown that the traditional opinion that the algebraic approach only works for context free languages is incorrect, and that the incorporation of a new formalism for synthesized or derived attributes is not necessary.

13 Towards a general theory of translation

In previous sections it was shown that translations arise between several kinds of languages. We have seen that in most fields the algebraic method was put forward, often independently of the proposals in other fields. The publications in the different fields discuss the same issues and use related notions. So, there seems to be a common basis for a general theory of translation. Below a first, small step will be made.

We may start with a principle for translating that can be seen as the philosophical background for the algebraic approach:

> **The principle of compositionality of translation**
> The translation of a compound expression is a function of the translations of its parts and of the rule by which the parts are combined.

The first formulation of this principle was in a publication concerning the machine translation project Eurotra, but the idea behind the principle can be found in older publications. A stronger (symmetric) form of the principle is the leading principle of the machine translation project Rosetta (Rosetta 1994). The principle is inspired by Frege's well known principle of compositionality of meaning. The formulation given above mirrors the formulation of Frege's principle in Partee, ter Meulen & Wall (1990, p. 318).

The principle of compositionality of translation can be formalized by requiring that source and target language are algebras SL and TL respectively, and that translating is a homomorphism between the term algebras T_{SL} to T_{TL}. However, for a practical reason, the definition below has more components. When two languages are given, they usually have their own internal structure and differ that much, that they do not have a similar syntax. Therefore the range of the translation function has to be an algebra that is constructed by means of polynomial operators available in the target algebra. This leads to the following definition.

Definition 25. Let the source language be defined by the algebra SL , and the target language by an algebra TL. Let δ be a polynomial derivor that transforms TL in an algebra similar to SL. Then a *compositional translation* from source language to target language is a homomorphism from the term algebra T_{SL} to the term algebra $T_{\delta(TL)}$. The translation is *correct* if there is a mapping $Decode$: $Sem_{TL} \rightarrow Sem_{SL}$ such that the restriction Dec of $Decode$ to $Sem_{\delta(TL)}$ is a homomorphism that makes the leftmost square in diagram (14) commute.
□

$$\begin{array}{ccccc}
T_{SL} & \xrightarrow{Tr} & T_{\delta(TL)} & \overset{\delta}{\longleftarrow\cdots\cdots} & Syn_{TL} \\
\downarrow Int_{SL} & & \downarrow Int_{\delta(TL)} & & \downarrow Int_{TL} \\
Sem_{SL} & \xleftarrow{Dec} & Sem_{\delta(TL)} & \overset{\delta}{\longleftarrow\cdots\cdots} & Sem_{TL} \\
 & \multicolumn{3}{c}{\underleftarrow{\qquad\qquad Decode \qquad\qquad}} &
\end{array}$$

Diagram 14: A general framework for compositional translation

Below we consider the components of this diagram and their relation with the articles we have discussed before.

* T_{SL}
  The source language is an algebra SL. If the expressions of the source language are ambiguous, SL is not suitable as domain of the translation homomorphism. Therefore the term algebra T_{SL} is used as domain for the translation homomorphism. Such ambiguities arise for natural languages (see sections 10 and 11). Ambiguities also arise for subexpressions of programming languages (see section 10). If the (sub)expressions of a fragment are not ambiguous, then the term algebra is isomorphic with the original algebra, and the original one can be used. This situation arises for logic, and for the examples used in articles about compiler construction. Therefore most authors use the original algebra as domain of the translation, and not the term algebra.

* $T_{\delta(TL)}$ and $Sem_{\delta(TL)}$
  The image of the translation homomorphism has to be an algebra similar to the source language algebra (otherwise it cannot be a homomorphism). Therefore a reconstruction of T_{SL} has to be made. The term 'embedding' from logic reflects this aspect, and the term 'reconstruction' is used frequently in Rus (1995). Related diagrams with derivors can be found in Tofte (1990). Other authors on algebraic compiler construction do not mention this aspect explicitly, although they proceed in the same way.

* δ
  The new operators needed to form a reconstruction of TL are obtained by polynomials. In the field of natural language this idea is introduced by Montague (1973). The role of polynomials is mostly not explicit in the field of

compiler construction, but polynomials are frequently used there. In translations of logics, polynomial translations are standard, but also non-polynomial translations are used sometimes, see section 9. In Rosetta there is, due to the design of the algebras, a direct correspondence of operators; but if one investigates the details of the operators, it is possible to view them as polynomials as well.

∗ *Dec* and *Decode*
If *Dec* exists, then it is unique, because T_{SL} is an initial algebra and the diagram has to commute. Then the image of $x \in Sem_{\delta(TL)}$ can be defined as $X = Int_{SL}(Tr^{-1}({Int_{TL}}^{-1}(x)))$; it only has to be checked that X is a singleton. However, in general, it is difficult to say what X is, because $Sem_{\delta(TL)}$ is not independently given, but defined indirectly (viz. by means of the translation and interpretation). Therefore, there is no information whether for $x \in Sem_{TL}$ also $x \in Sem_{\delta(TL)}$. This explains why usually not *Dec* is defined, but *Decode*; a function with the original meanings as domain; *Dec* is then its restriction to $Sem_{\delta(TL)}$. The ideal that there is an isomorphism between source meanings and target meanings (see section 2) can be reached by switching to a quotient algebra: $Sem_{\delta(TL)}/_{Ker(Dec)}$. Without this abstraction *Dec* will be an isomorphism only the exceptional case where the algebras are designed with this purpose (Rosetta, section 10).

Definition (25), the structure in diagram (14) and the comments given above, are just the first steps toward a general framework for translating. The following points require further research, and probably other issues as well.

1. Other translations
Although this study brings together a lot of algebraic translations, there certainly are more (for instance, from programming language to logic). The work on 'institutions' (connections between specification formalisms) deals with a related subject (Goguen & Burstall 1992). A further comparison may give rise to other questions and answers concerning algebraic translation.

2. Algebra
In all publications concerning programming languages many sorted algebras are used. For Rosetta a one sorted algebra is used, and this also is the case in Montague grammar (see Janssen (1986) for a many sorted version). However, for applications to natural language order sorted algebras seem most appropriate, and maybe this is also the case for programming languages (Goguen & Malcolm 1996).

3. Homomorphism
Most publications follow the standard definition of a homomorphism for many sorted algebras. Rus (1991) argues for generalized homomorphisms: mappings which may change the signature (the sort structure). Rosetta (1994, p. 393) gives another generalization: homomorphisms which have not only elements in their range, but also operators. These homomorphisms may not only map two elements to one image, but also two operators.

4. Tools
   In projects which deal with larger fragments of language, tools are needed in order to perform all the tasks in an algebraic way. Some of the publications mentioned in this article describe such projects which have developed tools: (Rosetta 1994, Rus 1995, Tofte 1990), another is Müller-Olm (1997).

5. Meanings
   In the examples we have considered the semantic models are used in two distinct ways. One is that the whole set of models is essential, the other that the interpretation is with respect to an intended model (not necessarily a fixed one). Logic and natural language are of the first kind, compiler construction is mostly of the other kind. The relation between the two uses of models needs clarification (see section 9 and 6).

6. Properties
   Some of the papers discussed here, are of a theoretical nature, and prove properties about the framework, e.g. Shapiro (1991) and Rosetta (1994, ch. 19). These results might find their place in a coherent framework.

14 Conclusion

In this article we have seen many examples of translations, and sketched a common, algebraic, framework. Even the notion 'correct translation' turned out to be closely related in all fields. Inspired by this unity, I would like to conclude with a quotation from 'Universal Grammar' (Montague (1970, p. 313), reprinted in Thomason (1974, p. 222)), in which I made two adaptations (indicated in italics):

> There is in my opinion no important theoretical difference between natural languages and the artificial languages of logicians *and computer scientists*; indeed I consider it possible to comprehend *all these* kinds of languages within a single natural and precise mathematical theory.

Acknowledgments

I thank the following persons for their useful remarks or assistance during the preparation of this article: Mark van de Brand, Peter van Emde Boas, Dick Grune, Lex Hendrix, Dick de Jongh, Karen Kwast, Teodor Rus, Martin Steffens, Albert Visser and two anonymous referees. I especially thank Anton Nijholt for his inducement to write this interdisciplinary study and Guiseppe Scollo for his many remarks on formulation and content.

References

Bancilhon, F. & Spyratos, N. (1981), 'Update semantics of relational views', *ACM Transactions on data bases* **6**, 557–575.

Burstall, R.M. & Landin, P.J. (1969), Programs and their proofs: an algebraic approach, *in* B. Meltzer & D. Michie, eds, 'Machine Intelligence', Vol. 5, American Elsevier, New York, chapter 2, pp. 17–43.

Chirica, L.M. (1976), Contributions to compiler correctness, PhD thesis, Dept. of Computer Science, University of California, Los Angeles. report UCLA-ENG-7697.

van Dalen, D. (1986), Intuitionistic logic, *in* D. Gabbay & F. Guenthner, eds, 'Handbook of philosphical logic. Vol III. Alternatives to classical logic', number 166 *in* 'Synthese Library', Reidel. Dodrecht, chapter 4, pp. 225–339.

Davidson, D. & Harman, G., eds (1972), *Semantics of natural language*, number 40 *in* 'Synthese library', Reidel, Dordrecht.

Dybjer, P. (1985), Using domain algebras to prove the correctness of a compiler, *in* K. Mehlhorn, ed., 'STACS 85, 2nd annual symposium on theoretical aspects of computer science, Saarbrücken, 1985', number 182 *in* 'Lecture notes in computer science', Springer, Berlin, pp. 98–108.

Epstein, R.L. (1990), *The semantic foundation of logic. vol 1. Propositional logic.*, number 35 *in* 'Nijhoff international philosophy series', Nijhoff/Kluwer, Dordrecht. Second edition published by Oxford University Press, Oxford.

Frege, G. (1923), Logische Untersuchungen. Dritter Teil: Gedankenfüge, *in* 'Beiträge zur Philosophie des Deutschen Idealismus', Vol. III, pp. 36–51. Reprinted in I. Angelelli (ed.), *Gottlob Frege. Kleine Schriften*, Georg Olms, Hildeheim, 1967, pp. 378-394. Translated as *Compound thoughts* in P.T.Geach & R.H. Stoothoff (transl.), *Logical investigations. Gottlob Frege*, Basil Blackwell, Oxford, 1977, pp. 55-78.

Goguen, J. & Burstall, R. (1992), 'Institutions: abstract model theory for specification and programming', *Journal of the Association for Computing Machinery* **39**(1), 95–146.

Goguen, J.A. & Malcolm, G. (1996), *Algebraic semantics of imperative programs*, Foundations of computing, The MIT Press, Cambridge, Mass.

Goguen, J.A., Thatcher, J.W., Wagner, E.G. & Wright, J.B. (1977), 'Initial algebra semantics and continuous algebras', *Journal of the Association for Computing Machinery* **24**, 68–95.

Groenendijk, J. & Stokhof, M. (1982), 'Semantic analysis of wh-complements', *Linguistics and Philosophy* **5**, 175–233.

Janssen, T.M.V. (1986), *Foundations and Applications of Montague Grammar: part 1, Philosophy, Framework, Computer Science*, number 19 *in* 'CWI tract', Centre for Mathematics and Computer Science, Amsterdam.

Janssen, T.M.V. (1997), Compositionality (with an appendix by B. Partee), *in* J. van Benthem & A. ter Meulen, eds, 'Handbook of logic and language', Elsevier, Amsterdam and The MIT Press, Cambridge, Mass., chapter 7, pp. 417–473.

Janssen, T.M.V. (1998), A survey of compositional translations, *in* W-P de Roever, H.Langmaack & A. Pnueli, eds, 'Int. Symposium , COMPOS'97, Bad Malente, Germany, 1997', number 1536 *in* 'Lecture Notes in computer Science', Springer, Berlin, pp. 327–349.

Knuth, D.A. (1968), 'Semantics of context-free languages', *Mathematical systems theory* **2**, 127–145.

Landsbergen, J. (1981), Adaption of Montague grammar to the requirements of parsing, *in* J.A.G. Groenendijk, T.M.V. Janssen & M.B.J. Stokhof, eds, 'Formal methods in the study of language. Proceedings of the third Amsterdam colloquium', number 136 *in* 'CWI Tracts', Centre for Mathematics and Computer Science, Amsterdam, pp. 399–420.

Lewis, D. (1970), 'General semantics', *Synthese* **22**, 18–67. Reprinted in Davidson & Harman (1972, pp. 169-248) and in Partee (1976, pp. 1-50).

van der Meer, H. (1994), Syllabus: Inleiding informatica, Dept. of Comp.Sc., University of Amsterdam.

Meijer, E. (1992), Calculating compilers, PhD thesis, Katholieke University Nijmegen, Nijmegen. ISBN 90-9004673-9.

Milner, R. & Weyrauch, R. (1972), Proving compiler correctness in a mechanized logic, *in* B. Meltzer & D. Michie, eds, 'Machine Intelligence', Vol. 7, American Elsevier, New York, chapter 3, pp. 51–70.

Montague, R. (1970), 'Universal grammar', *Theoria* **36**, 373–398. Reprinted in Thomason (1974, pp. 222-246).

Montague, R. (1973), The proper treatment of quantification in ordinary English, *in* K.J.J. Hintikka, J.M.E. Moravcsik & P. Suppes, eds, 'Approaches to natural language, Synthese Library 49', Reidel, Dordrecht, pp. 221–242. Reprinted in Thomason (1974, pp. 247-270).

Morris, F.L. (1973), Advice on structuring compilers and proving them correct, *in* 'Proceedings ACM Symposium on principles of programming languages, Boston, 1973', Association for Computing Machinery, pp. 144–152.

Moscowitz, Y. & Shapiro, E. (1993), On the structural simplicity of machines and languages, Technical Report CS93-04, Weizman institute, dept. of appl. math. and comp. sc., Rehovot, Israel. to appear in 'Annals of mathematics and artificial intelligence'.

Mosses, P. (1980*a*), A constructive approach to compiler correctness, *in* N.D. Jones, ed., 'Semantics-directed compiler generation (Proc. Workshop, Aarhus)', number 94 *in* 'Lecture notes in computer science', Springer, Berlin, pp. 189–210.

Mosses, P. (1980*b*), A constructive approach to compiler correctness, *in* J. de Bakker & J. van Leeuwen, eds, 'Automata, languages and programming (seventh colloquium, Noordwijkerhout)', number 85 *in* 'Lecture notes in computer science', Springer, Berlin, pp. 449–469.

Müller-Olm, M. (1997), *Modular Compiler verification*, number 1283 *in* 'Lecture notes in computer science', Springer, Berlin.

Partee, B., ed. (1976), *Montague grammar*, Academic Press, New York.

Partee, B., ter Meulen, A. & Wall, R.E. (1990), *Mathematical Methods in Linguistics*, number 30 *in* 'Studies in Linguistics and Philosophy', Kluwer, Dordrecht.

Polak, W. (1981), *Compiler specification and verification*, number 124 *in* 'Lecture notes in computer science', Springer, Berlin.

Pullum, G.K. & Gazdar, G. (1982), 'Natural languages and context-free languages', *Linguistics and Philosphy* **4**, 471–504. reprinted in Savitch et al. (1987, pp.138–182).

Rosetta, M.T. (1994), *Compositional Translation*, number 230 *in* 'The Kluwer International Series in Engineering and Computer Science', Kluwer, Dordrecht. (M.T. Rosetta = {L. Appelo, T. Janssen, F. de Jong, J. Landsbergen (eds)}).

Royer, V. (1986), Transformations of denotational semantics in semantics directed compiler generation, *in* 'Proceedings of the SIGPLAN '86 symposium on compiler construction', SIGPLAN notices Vol. 21, Nr. 7, ACM (Association for Computing Machinery), pp. 68–73.

Rus, T. (1976), Context-free algebra: a mathematical device for compiler specification, *in* A. Mazurkiewicz, ed., 'Mathematical foundations of computer science 1976 (5th symp. Gdansk)', number 45 *in* 'Lecture notes in computer science', Springer, Berlin, pp. 488–494.

Rus, T. (1980), 'Has-hierarchy; a natural tool for language specification', *Fundamenta Informaticae* **3**, 269–294.

Rus, T. (1987), 'An algebraic model for programming languages', *Computer Languages* **12**, 173–195.

Rus, T. (1991), 'Algebraic construction of compilers', *Theoretical Computer Science* **90**, 271–308.

Rus, T. (1995), Algebraic processing of programming languages, *in* A. Nijholt, G. Scollo & R. Steetskamp, eds, 'Algebraic methods in language processing AMILP'95', number 10 *in* 'Twente workshop In language technology', Universiteit Twente, Enschede, pp. 1–41.

Rus, T. & Halverson, T. (1994), 'Algebraic tools for language processing', *Computer Languages* **20**, 213–238.

Savitch, W.J., Bach, E., Marsh, W. & Safran-Naveh, G. (1987), *The formal complexity of natural language*, number 33 *in* 'Studies in Linguistics and Philosphy', Reidel, Dordrecht.

Shapiro, E. (1991), Separating concurrent languages with categories of language embeddings, *in* 'Proceedings of the 23 annual symposium on theory of computing (STOC'91, New Orleans)', ACM, pp. 198–208.

Shapiro, E. (1992), Embeddings among concurrent programming languages, *in* W. R Cleaveland, ed., 'Proc. third international conference on concurrency theory, Stony Brook 92', Springer lecture notes in computer science, Springer, Berlin, pp. 486–503.

Tanenbaum, A. S. (1975), A critique of the CDC cyber computers, IR 5, Wiskundig Seminarium, Vrije Universiteit, Amsterdam.

Tanenbaum, A. S. (1976), *Structured computer organization*, Prentice-Hall.

Thatcher, J.W., Wagner, E.G. & Wright, J.B. (1979), More on advice on structuring compilers and proving them correct, *in* H.A. Maurer, ed., 'Automata, languages and programming. (Proc. 6th. coll. Graz)', number 71 *in* 'Lecture notes in computer science', Springer, Berlin.

Thomason, R.H. (1974), *Formal Philosophy. Selected Papers of Richard Montague*, Yale University Press, New Haven.

Tofte, M. (1990), *Compiler generators. What they can do, what they might do, and what they will probably never do*, number 19 *in* 'EATCS monographs on theoretical computer science', Springer, Berlin.