Science of Computer Programming 3 (1983) 223-278
North-Holland

INCREASING MODULARITY AND LANGUAGE-INDEPENDENCY IN AUTOMATICALLY GENERATED COMPILERS*

Harald GANZINGER
Institut für Informatik, Technische Universität München, D-8000 München 2, Fed. Rep. Germany

Communicated by M. Paul
Received June 1983

1. Introduction

The aim of this paper is to introduce a method for obtaining modular compiler descriptions that, paraphrazing [26],
- exhibit a semantic processing based on fundamental concepts of languages and compiling;
- are easily modifyable and adaptable to different but related languages;
- are combinations of language-independent modules;
- are subject to automatic compiler generation.

1.1. Related work

The work reported here is based on ideas from (modular) algebraic specifications of abstract data types [31, 2, 6, 23], abstract semantic algebras [25, 26], and compiler descriptions based on attribute grammars [22].

Many papers have utilized ideas of abstract data type theory to improve the structure of semantics definitions and/or compiler descriptions. In [4], following [10] and [24], the fundamental algebraic structure of denotational semantics definitions and syntax-oriented compiler descriptions has been recognized. Following [30], in [7], [16], and [20] the use of abstract data types has been suggested. In particular the latter paper was concerned with structuring compiler definitions hierarchically, using the specification language OBJ [19]. In addition, many authors of denotational descriptions have tried to impose structure on their descriptions. In particular [29] and [30] proposed general language independent combinators, abbreviating pieces of λ-notation. Algebraic descriptions of compilers in the context of automatic compiler generation are considered in [16] and [8]. Modularization of compiler descriptions is not investigated in these papers.

* This work was in part supported by the Sonderforschungsbereich 49, Programmiertechnik, at the Technical University of Munich.

We agree with [26] in that none of the mentioned approaches has succeeded in reaching the goals that have been stated above.[1] A detailed justification of this claim is to be found in [26]. We give the intuitive reason for the principal problem that arises in semantics and compiler descriptions.

1.2. The problem

The theory of abstract data types in the classical sense [21, 31, 2] views a software module as a package of functions which the user of the module may call. The data elements themselves are only implicitly given by the set of all terms in the functions.

Compiler modules decorate nodes of parse trees with semantic information. E.g., a module that handles the declarations associates declaration information with applied occurrences of identifiers. So, it has to handle data of sorts

$$\mathit{Stat},\ \mathit{Id},\ \mathit{DeclInfo}$$

and to provide a function

$$\mathit{find}: \mathit{Stat},\ \mathit{Id} \to \mathit{DeclInfo},$$

where *Stat* represents all statement nodes in a parse tree. $\mathit{find}(s, x)$ is supposed to find that declaration for x that is visible at s. Specifying the properties of the elements of *Stat* requires to model this set of nodes together with their syntactic relationships in the program. E.g., it has to be specified that

$$\mathit{find}(s, x) = d$$

if s is a statement in a scope p containing a declaration c that binds the identifier x to d. Thus, it is not sufficient to know the internal structure of a statement s. The context of s in the program is relevant, too.

Authors of algebraic specifications of languages and compilers usually consider the syntax of a language to be given by a system of (syntactic) operators, e.g.

$$\begin{array}{ll} \textbf{if } \mathit{Exp} \textbf{ then } \mathit{Stat} \textbf{ else } \mathit{Stat} & \to \mathit{Stat} \\ \mathit{Var} := \mathit{Exp} & \to \mathit{Stat} \\ \dots & \end{array}$$

In any model, *Stat* is, then, the set of all objects that can be represented as terms in *if_then_else_*, _:=_, etc. The context in which a syntactic construct occurs in a program is not available. As a consequence, this technique is not adequate to model an algebra of nodes in parse trees. Rather, something that establishes the following equation would be needed:

$$\mathit{Stat} = \{(t, v) \mid t \text{ parse tree, } v \text{ node in } t \text{ labelled by } \mathit{Stat}\}$$

Now, to generate this set would require to specify primitive functions such as $\mathit{son}(t, v)$ and $\mathit{ancestor}(t, v)$. These primitives are no total functions. It is well known

[1] Some of the mentioned papers are concerned with semantics descriptions so that only goals (i)–(iii) apply.

that algebraic specifications of partial functions tend to be complicated. Even if this was acceptable, the level on which specifications would have to be written is too low. To specify the detection of syntactic patterns in order to decide upon which semantic rule to apply has to be 'implemented' by hand in terms of tree-walk operations.

Mosses [25, 26], circumvents this difficulty by indexing semantic operators, such as *find* in the above example, by semantic information about the syntactic context in which it is applied. This goes beyond the classical technical and, as we believe, methodological framework of abstract data type specifications. Mosses' specifications are two-levelled: One level provides the specification of the index algebras and a second level contains the specification of the properties of the semantic operators. The mathematical basis for specifications of this kind is in this author's view not fully developed yet. Moreover, Mosses does not yet provide a formal basis for combining his 'semantic algebras'. (Such a framework exists for specifications in the classical sense [6, 3, 5, 23].) In [26], it is still in general necessary to modify the axioms of one semantic algebra to be able to combine it with a second algebra. It is not at all clear, how language specifications can be obtained as combinations of the specified language concepts. Nevertheless, Mosses' approach has motivated a great deal of this work. To get around the problems that exist with Mosses' approach, we have developed a different extension of the classical concepts.

1.3. *Basic idea*

We represent syntactic conditions such as

s is a statement in a scope p

or

c is a declaration in a scope p

as *relations* in an appropriate algebraic *structure*. The specification of a compiler module, thus, consists of the specification of two kinds of objects:
- relations between syntactic constructs,
- functions that yield semantic values denoted by syntactic constructs.

Consider, again, a module that implements symbol table handling for a PASCAL-like language. Here, the relevant language constructs are declarations, scopes, and applied occurrences of identifiers. Syntactic relations between such constructs can be expressed as formulas in the following four elementary relations:

$$\begin{aligned}
r1(C, C') &\equiv C \rightarrow \text{scope with binding constructs } C',\\
r2(C, C1, C2) &\equiv C \rightarrow C1;C2,\\
r3(C) &\equiv C \rightarrow \text{noBindings}\\
r4(C, I, D) &\equiv C \rightarrow \text{bind } I \text{ to } D.
\end{aligned}$$

The relation symbols define a 'language' of constructs c that can be viewed as an *abstraction* of the concrete language PASCAL. This abstraction is specifically tailored to the language facet 'binding of identifiers'. The abstract syntactic notions are scopes and declarations. Any language facet (e.g. statements) that is irrelevant with respect to this problem is only referred to as a construct that contains no bindings. The semantic objects which the module compiles are represented by operators. In our example we have, e.g.,

$$find(C, I).$$

The abstract semantic notions are identifiers I, declaration information D, and the finding of declarations to which an identifier is bound to. The latter depends on the context C of binding constructs of the applied occurrence of the identifier I.

For any concrete program, the identification process itself is viewed as the evaluation of *find* at (syntactic) constructs C. The latter are described as solutions to formulas built from the relation symbols. The formulas represent the abstract structure of the context of C in a program. E.g., the fact that C is an applied occurrence of an identifier x that is directly contained in a scope S which introduces exactly one declaration of a variable x of type integer would, then, be expressed by the formula

$$(S \rightarrow \text{scope with binding constructs } C1) \wedge (C \rightarrow \text{noBindings})$$
$$\wedge (C1 \rightarrow C2; C) \wedge (C2 \rightarrow \text{bind } x \text{ to 'integer'})$$

where S, $C1$, $C2$, and C are variables. Now, the symbol table handling module would have to return $find(c, x) = integer$, for any value c for C that makes the above formula to be satisfyable.

Our approach will be based on the following basic conceptual assumptions about compiler modules:

(1) The specification of (the implementation of) a compiler module consists of the definition of a set of new types, functions, and relations from given types, functions, and relations.

(2) The relations represent syntactic facts about the context of a construct in a program. The functions evaluate semantic information.

(3) The user of a compiler module may ask the evaluation of semantic operators applied to syntactic constructs where the latter are identified as *solutions to formulas* built from relations. This goes beyond the classical framework of imperative or functional programming. Not the characteristic function of a relation is called by the user (this is how relations are used in imperative and functional programs). Rather, the set which the relation represents (e.g. set of tuples $(S, C, C1, C2)$ that satisfies the above formula) is the data object that is to be explicitly manipulated by the module. (In language implementations, the parse tree of a program is a data structure in which all syntactic relations can be represented.) Modules in this sense are related to nonprocedural or relational programming in languages such as PROLOG.

2. Representation of modules

2.1. Informal introduction

The *signature* of a module lists the sets of sorts (i.e. type names), operators (i.e. function symbols), and relation symbols, together with their parameter and result sorts, which the module exports.

Example 2.1. Signature of module *Alloc*.

$*Integer, MemUnit*

---

size(*MemUnit*) : *Integer*
offset(*MemUnit*) : *Integer*
address(*MemUnit*) : *Integer*

---

memory → segment consisting of *MemUnit*
MemUnit → segment consisting of *MemUnit*
MemUnit → elementary of size *Integer*
MemUnit → *MemUnit* concatenated *MemUnit*
MemUnit → *MemUnit* overlapped *MemUnit*

The example gives the signature of a compiler module that performs storage allocation for source program data. In the examples, "---" will be used to separate sorts, operators, and relation symbols. Sorts are named by identifiers starting with an upper-case letter. Operators will be written in either prefix or mixfix notation. Relation symbols will be written as 'syntactic productions'

$$\tilde{}\, Y_1 \tilde{} \cdots \tilde{} Y_m \tilde{} \rightarrow \tilde{} X_1 \tilde{} \cdots \tilde{} X_n \tilde{}, \qquad m, n \geq 0,$$
X_i, Y_i sorts.

(Here, "$\tilde{}$" stands for strings that are, like "→", parts of the name of the relation.) Production rules define the kind of syntactic derivation relations which the relation symbols are supposed to capture. Note, however, that we do not employ the concept of a grammar nor do we restrict ourselves to context-free rules. In principle, a relation symbol denotes an arbitrary set of signature

$$Y_1 \times \cdots \times Y_m \times X_1 \times \cdots \times X_n.$$

E.g., *MemUnit* → elementary of size *Integer* represents binary relations

$$\_ \rightarrow \text{elementary of size}\_ \subseteq \mathit{MemUnit} \times \mathit{Integer}.$$

Similarly, memory → segment consisting of *MemUnit* is a unary relation

$$\text{memory} \rightarrow \text{segment consisting of}\ \_ \subseteq \mathit{MemUnit}.$$

Formally, *signatures* $\Sigma = (S, \Omega, R)$ consist of a set of sorts S, an $S^* \times S$-indexed family of sets $\Omega_{s_1 \cdots s_n s_0}$ of operators, and an S^*-indexed family of sets $R_{s_1 \cdots s_n}$ of relation symbols. Operators f with parameter sorts $s_1, \cdots, s_n$ and result sort s_0 are denoted by $f:s_1 \cdots s_n s_0$.[2] Similarly, relation symbols r with argument sorts $s_1, \cdots, s_n$ are written as $r:s_1 \cdots s_n$.

Generally, if $Y=(Y_i)_{i \in I}$ is a family of sets, then we will also denote by Y the disjoint union $\{y:i \mid i \in I, y \in Y_i\}$ of the Y_i. We will omit the index $:i$ if no confusion can arise.

The representation of a module is established by defining its sorts, operators, and relation symbols in terms of already defined ones.

The *representation* of a (compiler) *module M* is defined by a signature morphism, called *representation (map)*, $\rho:\Sigma_M \to \Sigma_I$, where Σ_M is the signature of the module M and Σ_I is the signature of types, operators, and relation symbols which the module *imports*.

Signature morphisms will be formally introduced below. We will first illustrate their intuitive meaning as specification of the representation (or implementation) of the components of a module. ρ defines the sorts, operators, and relation symbols of Σ_M in terms of the Σ_I-constructs.

Example 2.2. Module *Alloc*: Signature and representation.

Integer ↦ *Integer*; λ SIZE
MemUnit ↦ (*iTop, sTop, offset, size*: *Integer*);
λ UNIT, UNIT1, UNIT2, UNIT3, U
---
size(U):*Integer* ↦ U.*size*
offset(U):*Integer* ↦ U.*offset*
address(U):*Integer* ↦ U.*iTop*
---
memory → segment consisting of UNIT ↦
UNIT.*iTop* = 0
UNIT.*offset* = 0
UNIT → segment consisting of UNIT1 ↦
UNIT1.*iTop* = UNIT.*iTop*
UNIT.*size* = UNIT1.*size*
UNIT1.*offset* = 0
UNIT.*sTop* = UNIT1.*sTop*
UNIT → elementary of size SIZE ↦
UNIT.*sTop* = UNIT.*iTop* + SIZE
UNIT.*size* = SIZE
UNIT1 → UNIT2 concatened UNIT3 ↦
UNIT2.*offset* = UNIT1.*offset*
UNIT3.*offset* = UNIT1.*offset* + UNIT2.*size*
UNIT2.*iTop* = UNIT1.*iTop*
UNIT3.*iTop* = UNIT2.*sTop*

[2] In examples we will also use the notation $f(s_1, \ldots, s_n):s_0$ or $f:s_1 \cdots s_n \to s_0$.

$$\text{UNIT1}.sTop = \text{UNIT3}.sTop$$
$$\text{UNIT1}.size = \text{UNIT2}.size + \text{UNIT3}.size$$
$$\text{UNIT1} \to \text{UNIT2 overlapped UNIT3} \mapsto$$
$$\text{UNIT2}.offset = \text{UNIT1}.offset$$
$$\text{UNIT3}.offset = \text{UNIT1}.offset$$
$$\text{UNIT2}.iTop = \text{UNIT1}.iTop$$
$$\text{UNIT3}.iTop = \text{UNIT1}.iTop$$
$$\text{UNIT1}.sTop = max(\text{UNIT2}.sTop, \text{UNIT3}.sTop)$$
$$\text{UNIT1}.size = max(\text{UNIT2}.size, \text{UNIT3}.size)$$

The example gives the representation map for the module *Alloc*

$$\rho_A : \Sigma_{Alloc} \to \Sigma_{Standard}$$

where $\Sigma_{Standard}$ is the signature of predefined standard types that include a definition of *Integer*. For the intuitive meaning behind these definitions cf. next section. For now we are only interested in the constituents and the basic form of the representation maps we are dealing with.

We have used the notation

$$x \mapsto \rho(x)$$

for sorts, operators, and relation symbols x. This means that $\rho(x)$ is the representation of x. For sorts s, we allow the representation by tuples of imported sorts. This tupling is what would be called a record type in a PASCAL-like language. E.g.

$$MemUnit \mapsto (iTop, sTop, offset, size : Integer);$$
$$\lambda \text{ UNIT, UNIT1, UNIT2, UNIT3, U}$$

means that the carrier set for sort *MemUnit* is represented by the cartesian product

$$Integer \times Integer \times Integer \times Integer$$

where *iTop*, *sTop*, *offset*, and size are the names of the projections to the single components, e.g. if $\text{U} \in MemUnit$, then $\text{U}.offset$ is the third component of U. λ UNIT, ..., U declares UNIT, ..., U to be variables of sort *MemUnit* to be used in the definition of the representation of the operators and relation symbols.

$$Integer \mapsto Integer$$

specifies that integers are represented by themselves.

Operators are represented as tuples in terms of imported operators. Terms may also contain projections to components of variables. E.g.

$$offset(\text{U}) \mapsto \text{U}.offset$$

specifies that the offset of a *MemUnit* U is its offset component. Similarly, the address of U is its *iTop*-component. These are very simple examples for representations of functions. A more complicated example would be

$$shift(\text{U}) \mapsto (\text{U}.iTop + 1, \text{U}.sTop + 1, \text{U}.offset + 1, \text{U}.size)$$

if *shift* was an operator of signature *shift*(*MemUnit*):*MemUnit*.

Needless to say that the operator representations have to be consistent with the sort representations in the following sense: if $f: s_1 \cdots s_n s_0$, then $\rho(f)$ is an $\rho(s_0)$-tuple of terms with a parameter sequence $\rho(s_1 \ldots s_n)$. So a more exact notation would be

λ U.*offset*(U) $\mapsto$ λ U.*iTop*, U.*sTop*, U.*offset*, U.*size*. (U.*offset*)
λ U.*shift*(U) $\mapsto$
λ U.*iTop*, U.*sTop*, U.offset, U.*size*. (U.*iTop* + 1, U.*sTop* + 1, U.*offset* + 1, U.*size*).

Relation symbols are represented by formulas over imported operators and relations. Here we restrict ourselves to formulas that are finite conjunctions of the following kinds of atomic formulas:
- equations of form $x0 = x1$ or $x0 = f(x1, \ldots, xn)$, with variables xi and operator f;
- relation expressions $r(x1, \ldots, xn)$, with a relation symbol r and variables xi.

For technical reasons, we have restricted ourselves to equations of the above types as the possibility of defining conjunctions of such equations allows for simulating any equation between arbitrary terms.[3] In the above example,

UNIT $\rightarrow$ elementary of size SIZE $\mapsto$
UNIT.*sTop* = UNIT.*iTop* + SIZE
UNIT.*size* = SIZE

specifies the relation _ $\rightarrow$ elementary of size_ as

(UNIT, SIZE) $\in$ _ $\rightarrow$ elementary of size_ $\Leftrightarrow$
UNIT.*sTop* = UNIT.*iTop* + SIZE $\wedge$ UNIT.*size* = SIZE.

Thus, we omit the $\wedge$-symbols between the atomic formulas of the conjunctions. Again, the relation symbol representation has to be consistent with the sort representation: if r is a relation symbol with parameter sorts $s_1 \cdots s_n$, then $\rho(r)$ is an expression that represents a predicate with parameter sorts $\rho(s_1) \cdots \rho(s_n)$. A more precise notation would, e.g., be

λ UNIT, SIZE. (UNIT $\rightarrow$ elementary of size SIZE) $\mapsto$
λ UNIT.*iTop*, UNIT.*sTop*, UNIT.*offset*, UNIT.*size*.
(UNIT.*sTop* = UNIT.*iTop* + SIZE $\wedge$ UNIT.*size* = SIZE).

Similarly,

λ UNIT1, UNIT2, UNIT3. (UNIT1 $\rightarrow$ UNIT2 overlapped UNIT3) $\mapsto$
λ UNIT1.*iTop*, UNIT1.*sTop*, UNIT1.*offset*, UNIT1.*size*,
UNIT2.*iTop*, UNIT2.*sTop*, UNIT2.*offset*, UNIT2.*size*,
UNIT3.*iTop*, UNIT3.*sTop*, UNIT3.*offset*, UNIT3.*size*.
(UNIT2.*offset* = UNIT1.*offset* $\wedge$ UNIT3.*offset* = UNIT1.*offset* $\wedge$
UNIT2.*iTop* = UNIT1.*iTop* $\wedge$ UNIT3.*iTop* = UNIT1.*iTop* $\wedge$

[3] In the examples we will deliberately drop some of these restrictions and also use terms as parameters of functions and relations.

$$\text{UNIT1}.sTop = max(\text{UNIT2}.sTop, \text{UNIT3}.sTop) \wedge$$
$$\text{UNIT1}.size = max(\text{UNIT2}.size, \text{UNIT3}.size))$$

is the more precise notation for the representation of the overlapping relation between memory units.

2.2. *Signature morphisms*

The formal model of the representation of a module, called *signature morphism*, is now being introduced. We start by listing some basic notions. Let X be an S-indexed family of sets (of *variables*). Furthermore, let $T_\Omega(X)$ be the free Ω-algebra over X, i.e. the set of all terms that can be written in the variables of X and the operators in Ω. $T_\Omega(X)_s$ is the set of term with result sort s. Then, for $u, v \in S^*$, $u = s_1 \cdots s_n$, $v = s'_1 \cdots s'_m$,

$$T_\Omega(u:v) = \{\lambda\, x.1, \ldots, x.n.\ (t_1, \ldots, t_m) \mid t_i \in T_\Omega(Y)_{s'_i}\},$$
$$\text{where } Y = \{x.i{:}s_i \mid i = 1, \ldots, n\}.$$

$T_\Omega(u{:}v)$ is the set of (tuples of) terms with parameter sequence of sort u and with a result of sort v.

Example 2.3.

$$\lambda\ \text{U}.iTop, \text{U}.sTop, \text{U}.offset, \text{U}.size.$$
$$(\text{U}.iTop + 1, \text{U}.sTop + 1, \text{U}.offset + 1, \text{U}.size)$$

is a term with parameters and results in

$$Integer \times Integer \times Integer \times Integer.$$

The set $F_\Sigma(X)$ of formulas over X is defined as

(1) $x{:}s, y{:}s \in X \Rightarrow x{:}s = y{:}s \in F_\Sigma(X)$,

(2) $f{:}s_1 \cdots s_n s_0 \in \Omega, x_i{:}s_i \in X \Rightarrow x_0{:}s_0 = f(x_1{:}s_1, \ldots, x_n{:}s_n) \in F_\Sigma(X)$,

(3) $r{:}s_1 \cdots s_n \in R, x_i{:}s_i \in X \Rightarrow r(x_1{:}s_1, \ldots, x_n{:}s_n) \in F_\Sigma(X)$,

(4) $q1, q2 \in F_\Sigma(X) \Rightarrow q1 \wedge q2 \in F_\Sigma(X)$.

Formulas become relation expressions by making some of their variables to be bound variables. Given $u = s_1 \cdots s_n \in S^*$, $E_\Sigma(u)$ is the set of *relation expressions*

$$E_\Sigma(u) = \{\lambda\, x_1{:}s_1, \ldots, x_n{:}s_n.\ q \mid q \in F_\Sigma(X).\ \text{for any } X \text{ that contains the } x_i{:}s_i\}.$$

The prefix λ makes the x_i to be the bound variables of q. The remaining variables in q are the free variables of q. For $Q = \lambda\, x_1, \ldots, x_n.\ q$, $Q(y_1, \ldots, y_n)$, y_i pairwise distinct, denotes the result of replacing in q any occurrence of the ith bound variable x_i by y_i. In what follows we will consider two relation expressions $Q1$, $Q2$ to be equal, if $Q1$ can be obtained from $Q2$ by consistently renaming its variables.

Definition 2.1. Given two signatures Σ and Σ', a signature morphism $\sigma:\Sigma\to\Sigma'$ consists of three components:
- a sort map $\sigma_S: S\to S'^{+}$ sending any sort s to a nonempty tuple $\sigma_S(s)$ of sorts,[4]
- a $S^*\times S$ indexed family of operator maps $\sigma_{\Omega_{us}}$ sending any operator $f\in\Omega_{us}$ with parameter sorts u and result sort s to a term $\sigma_{\Omega_{us}}(f)\in T_{\Omega'}(\sigma_S(u):\sigma_S(s))$,
- a S^*-indexed family of relation symbol maps σ_{R_u} sending any relation symbol $r\in R_u$ to a relation expression $\sigma_{R_u}(r)\in E_{\Sigma'}(\sigma_S(u))$.

To be able to compose signature morphisms, we extend σ to expressions $Q\in E_\Sigma(u)$ by

$$\begin{aligned}
\sigma(x_0=f(x_1,\ldots,x_n)) \;\equiv\;& \\
& x_0.1=g_1(x_1.1,\ldots,x_1.k_1,\ldots\;\;\ldots,x_n.1,\ldots,x_n.k_n)\\
& \quad\wedge\cdots\wedge\\
& x_0.k_0=g_k(x_1.1,\ldots,x_1.k_1,\ldots\;\;\ldots,x_n.1,\ldots,x_n.k_n)\\
& \quad\text{if } \sigma(f)=\lambda\, x_1,\ldots,x_n.\,(g_1,\ldots,g_k),\ x_j\equiv x_j{:}s_j, \text{ and} |\sigma(s_j)|=k_j,\\
\sigma(x_0=x_1) \;\equiv\;& x_0.1=x_1.1\wedge\cdots\wedge x_0.k_1=x_1.k_1,\\
\sigma(r(x_1,\ldots,x_n)) \;\equiv\;& \sigma(r)(x_1.1,\ldots,x_1.k_1,\ldots\;\;\ldots,x_n.1,\ldots,x_n.k_n),\\
\sigma(q1\wedge q2) \;\equiv\;& \sigma(q1)\wedge\sigma(q2).\\
\sigma(\lambda\, x_1,\ldots,x_n.\,q) \;\equiv\;& \lambda\, x_1.1,\ldots,x_n.k_n.\,\sigma(q).
\end{aligned}$$

In the above it is assumed that, if given σ and a variable x of sort s, then $x.i$ is a new variable of sort s'_i, if $\sigma(s)=s'_1\ldots s'_n$ and $1\leqslant i\leqslant n$.

Example 2.4. ρ_A sends

$A = \mathit{address}(\text{U})\wedge$
memory $\to$ segment consisting of U2 $\wedge$
U2 $\to$ U3 concatenated U $\wedge$
U3 $\to$ elementary of size 2

to

$A=\text{U}.\mathit{iTop}\wedge$
$\text{U2}.\mathit{iTop}=0\wedge \text{U2}.\mathit{offset}=0\wedge$
$\text{U3}.\mathit{offset}=\text{U2}.\mathit{offset}\wedge \text{U}.\mathit{offset}=\text{U2}.\mathit{offset}+\text{U3}.\mathit{size}\wedge$
$\text{U3}.\mathit{iTop}=\text{U2}.\mathit{iTop}\wedge \text{U}.\mathit{iTop}=\text{U3}.\mathit{sTop}\wedge \text{U2}.\mathit{sTop}=\text{U}.\mathit{sTop}\wedge$
$\text{U2}.\mathit{size}=\text{U3}.\mathit{size}+\text{U}.\mathit{size}\wedge$
$\text{U3}.\mathit{sTop}=\text{U3}.\mathit{iTop}+2\wedge \text{U3}.\mathit{size}=2$

[4] We do not allow sorts to be mapped to the empty sequence of sorts as this would later require to introduce operators with possibly empty result sequences. In principle, however, this restriction could be removed.

Theorem 2.1. *Signatures together with signature morphisms form a category*[5] *denoted* SIG.

The proof is obvious. The composition $\sigma = \sigma'\sigma''$ is defined by composing the sort, operator, an relation symbol maps, respectively.[6]

Semantically, signatures represent classes of algebraic structures. Signature morphisms define maps between such classes, thereby representing formally the process of implementing a module in terms of the constituents of pregiven modules.

Σ*-struct* is the class of Σ-structures together with Σ-homomorphisms between them. A Σ-structure A consists of (carrier) sets s_A, for any $s \in S$, of functions $f_A : s_{1_A} \times \cdots \times s_{n_A} \to s_{0_A}$, for any operator symbol $f: s_1 \cdots s_n s_0$, and of relations $r_A \subseteq s_{1_A} \times \cdots \times s_{n_A}$, for any relation symbol $r: s_1 \cdots s_n$. A Σ-homomorphism $h : A \to B$ between Σ-structures A and B is a S-sorted family of maps $h_s : s_A \to s_B$ for which

$$h_{s_0}(f_A(x_1, \ldots, x_n)) = f_B(h_{s_1}(x_1), \ldots, h_{s_n}(x_n)),$$

$$r_A(x_1, \ldots, x_n) \Rightarrow r_B(h_{s_1}(x_1), \ldots, h_{s_n}(x_n)),$$

for operators f and relation symbols r as above.

Semantically, relation expressions denote relations. Given a Σ-structure A and $Q \in E_\Sigma(u)$, $Q_A \subseteq u_A$ is defined as follows. If $Q \equiv \lambda\ x_1{:}s_1, \ldots, x_n{:}s_n.\ q$, then Q_A is the set of all $(a_1, \ldots, a_n)$ such that there exist values $(x{:}s)_A \in s_A$ for the variables $x{:}s$ in q such that $(x_i{:}s_i)_A = a_i$ and q becomes a valid assertion in A. For $u = s_1 \cdots s_n \in S^*$, we assume $u_A = s_{1_A} \times \cdots \times s_{n_A}$.

Theorem 2.2. *Let* $\sigma : \Sigma \to \Sigma'$ *be a signature morphism. Then there exists a functor* σ*-struct* : Σ'*-struct* $\to$ Σ*-struct such that the map that sends any signature* Σ *to* Σ*-struct and any signature morphism* σ *to* σ*-struct is a (contravariant) functor struct*: SIG $\to$ CAT, *where* CAT *is the category of all categories.*

Proof. Let $A' \in \Sigma'$*-struct.* We define σ*-struct*$(A') = A$ as follows. $s_A = \sigma(s)_{A'}$, i.e. the product of the A'-carriers of the sorts in $\sigma(s)$. For $f \in \Omega$, $f_A = g_{1_{A'}} \times \cdots \times g_{n_{A'}}$, if $\sigma(f) = (g_1, \ldots, g_n)$. For $r \in R$, $r_A = \sigma(r)_{A'}$. $\square$

In this paper, modules are assumed to be parameterized. Sorts, operators, and relation symbols are allowed as parameters of a module. Thus, the parameter is a sub-signature $\Sigma \subseteq \Sigma_M$. For conceptual simplicity we require the parameters of a module to be listed among the sorts, operators, and relation symbols, which the module imports. Thus, $\Sigma \subseteq \Sigma_I$ and the restriction of the representation map to Σ is the identity. The latter means that imported types must not be affected by the representation map. Moreover any standard type which the operators and relation symbols of the module refer to has to be listed among the parameters of the module.

[5] In this paper we assume the reader to be familiar with the basic definitions of a category and a functor.
[6] The composition of morphisms is written from right to left, i.e. $\sigma'\sigma''(x) = \sigma'(\sigma''(x))$.

This simplifies the technical treatment as it becomes unnecessary to distinguish between parameter types and predefined types. Then, the parameter of the module gives a complete specification about the required signature of possible environments. Thus, in the above example, *Integer* is considered a parameter of the module. This is indicated by the prefix $ which is used to distinguish parameter constructs. For the examples of this paper we can assume the following signature $\Sigma_{Standard}$ of standard types to be given:

Example 2.5. Standard types.

Integer, Bool

. . . all standard operators, e.g.

$+(Integer, Integer): Integer$

$=(Integer, Integer): Bool$

$max(Integer, Integer): Integer$

. . .

As stated above, $\Sigma_{Standard}$ has to be contained in any signature of a module so that, in the examples, we need not repeat its constituents.

The reader who is familiar with the use of attribute grammars will have noticed that in the definition of the module *Alloc*, the representation of relation symbols looks like an attribute grammar: Relation symbols r play the role of syntactic rules and their images under the representation map $\rho(r)$ are the attribute rules. The association of grammar symbols to attributes is captured by the sort representations. The attribute names are the names of the projections. In section 5 we will take a closer look at the correspondence of attribute grammars and signature morphisms. For now we simply state the following theorem.

Theorem 2.3. *Any attribute grammar is a signature morphism, i.e. specifies (the implementation of) a compiler module in the above sense.*

This observation, which was in fact a major goal of this research, has important practical consequences wrt. compiler generation, cf. Section 5. Note also, that the converse of the theorem is not true, i.e. our notion of compiler modules is more general than what is captured by attribute grammars. This will be the key to the kind of modularization which we have in mind.

3. The definition of some basic compiler modules

We give the definition of basic compiler modules for handling bindings, for allocation of memory for program data, and for code generation.

3.1. Binding identifiers to declarations

We assume the following PASCAL-like concepts of the visibility of declarations: Declarations occur in scopes that specify the region in which the declarations are visible. Scopes may contain inner scopes where identifiers can be redeclared. A scope must not contain more than one declaration of an identifier. At an application of an identifier two situations may occur: The identifier can be qualified, i.e. consist of an identification of a scope and the identifier itself. In this case, the corresponding declaration must be contained in the mentioned scope. (An example is the selection of a record field $x.s$ in PASCAL. Here, x denotes the record variable. The definition of its type is the scope in which the field names s are declared. That declaration is the defining occurrence for $x.s$.) Otherwise, if the identifier is not qualified, the declaration must be contained in a scope that is to be found in an enclosing scope. The declaration contained in the innermost such scope is, then, the defining occurrence.

The compiler module *Identification* is assumed to be implemented upon a given module *SymbolTable* that encapsulates operations on symbol tables. The signature of *SymbolTable* is given in Fig. 1. The sorts and operators are assumed to have the following properties. *StStates* is the domain of all states of the symbol table. *Id* is the domain of identifiers. Remember, the prefix S is supposed to indicate that *Id* is a (type-) parameter of the module, as is *DeclInfo*, the domain of semantic objects to which an identifier can be bound to. Integers are used to number scopes (e.g. record types, blocks). (Remember, standard types such as *Integer* and *Bool* together with their standard operators are considered as parameters if they are referred to by the module.) *init* initializes the symbol table. *enterScope* marks the begin of a new scope. *leaveScope* marks the end of a scope. *currentScope* returns the number of the current scope. *enter* enters a new declaration into the symbol table. *lookup* searches the symbol table for the declaration of the *Id*. *lookupQual* searches the

```
StStates, $Id, $DeclInfo[7]
---
$eq(Id, Id):Bool
init         :StStates
enterScope(StStates)              : StStates
leaveScope(StStates)              : StStates
currentScope(StStates)            : Integer
enter(StStates, Id, DeclInfo)     : StStates
lookup(StStates, Id)              : DeclInfo
lookupQual(StStates, Integer, Id) : DeclInfo
```

Fig. 1. Module "*SymbolTable*": Signature.

[7] Remember that we do not explicitly list the standard type sorts (in this case *Integer* and *Bool*) and operators that are required to be part of (the parameter) of any module signature.

symbol table for the declaration of the qualified identifier $x.s$. The last two operations can be assumed to employ the parameter equality predicate *eq* for comparing declaration entries in the symbol table with the identifier for which a declaration is looked after. *SymbolTable* does not provide relations, i.e. it is a module in the sense of functional and imperative programming. (Note that predicates such as *eq* have been specified as characteristic functions rather than relations as their characteristic function is the object of interest.) Therefore, we could have given a complete formal specification of this module in the style of [17] or [20]. This is, however, irrelevant for our considerations here. See Section 4.1. for a definition of a concrete module *SymbolTable*.

Figure 2 depicts the definition of the module *Identification*. It gives its signature $\Sigma_{Identification}$ as well as its representation, the signature morphism

$$\rho_I : \Sigma_{Identification} \to \Sigma_{SymbolTable},$$

that specifies its implementation over *SymbolTable*. The notation has been explained in Section 2. The intuitive meaning is as follows.

The bindings context *Bindings* of any program construct B is represented as a pair of symbol table states obtained during analysing B. *iSt* is the state before and *sSt* is the state after analysing the construct B.

$Integer $\mapsto$ Integer; λ I
$DeclInfo $\mapsto$ DeclInfo; λ DECLINFO
$Id $\mapsto$ Id; λ ID
Bindings $\mapsto$ (iSt:StStates, sSt:StStates); λ B, B′, B0, B1, B2
---
scope(B):Integer $\mapsto$ currentScope(B.iSt)
find(B, ID): $\mapsto$ lookup(B.iSt, ID)
findQual(B, I, ID):DeclInfo $\mapsto$ lookupQual(B.iSt, I, ID)
---
program → binding constructs B $\mapsto$
    B.iSt = init
B → scope with binding constructs B′ $\mapsto$
    B′.iSt = enterScope(B.iSt)
    B.sSt = leaveScope(B′.sSt)
B0 → B1;B2 $\mapsto$
    B1.iSt = B0.iSt
    B2.iSt = B1.sSt
    B0.sSt = B2.sSt
B → noBindings $\mapsto$
    B.sSt = B.iSt
B → bind ID to DECLINFO $\mapsto$
    B.sSt = enter(B.iSt, ID, DECLINFO)

Fig. 2. Module "Identification": Signature and implementation.

The operators of *Identification* are *scope*(_), *find*(_,_), and *findQual*(_,_,_). The current scope that directly encloses a binding construct b is obtained by applying the symbol table operation *currentScope* to the initial symbol table state of b. *find*(b, x) finds the declaration of the identifier x in the set b of bindings by evoking *lookup*. *findQual*(b, x, s) searches b for the bindings that have been established in the scope s. Then the declaration for x in s is returned.

The intuitive meaning of the relation symbols has already been illustrated in the last section. Their implementation using symbol table operations is as follows: At any scope construct, a new scope is opened (*enterScope*). The bindings B' that have been established inside the scope are hidden to the outside, i.e. made invisible among the set of all bindings B encountered so far (*leaveScope*). Sequences of bindings are processed from left to right. Any construct not containing a binding construct does not change the symbol table.

At this point it seems appropriate to recall the (meta-)semantic meaning of such definitions. Given a concrete module *SymbolTable*, i.e. an algebra A of signature $\Sigma_{SymbolTable}$, the relation _→scope with binding constructs_, for example, is in $A' = \rho_I\text{-}struct(A)$ defined as follows:

$$Bindings_{A'} = iSt{:}StStates_A \times sSt{:}StStates_A$$
$$\_\!\rightarrow \text{scope with binding constructs}\_{}_{A'} \subseteq Bindings_{A'} \times Bindings_{A'}, \text{ such that}$$
$$(B,B') \in \_\!\rightarrow \text{scope with binding constructs}\_{}_{A'} \Leftrightarrow$$
$$(B'.iSt = enterScope_A(B.iSt)) \wedge (B.sSt = leaveScope_A(B'.sSt))$$

Since *Id*, *DeclInfo*, *Bool*, and *Integer* are the parameters of *SymbolTable*, they are not the parameters of *Identification*, too.

3.2. Memory allocation

We assume that storage will be occupied by structured data and that the lifetimes of different data can be overlapping. E.g., variables in parallel blocks as well as different variants of records may be assigned to overlapping storage units. Figure 3 repeats signature and implementation of the module *Alloc* as already given in Section 2 based on standard arithmetic, i.e. as a signature morphism

$$\rho_A : \Sigma_{alloc} \rightarrow \Sigma_{Standard}.$$

Storage is assumed to be hierarchically structured into segments and units. A unit is either a sequential and/or overlapping structure of subunits, or it is a not further structured, elementary storage block, or it is again a segment. A memory unit is implemented by the memory top *iTop* before and the memory top *sTop* after allocating the unit. *size* is its size in terms of primitive units such as words or bytes. *address* is the (absolute) address of a unit in the program memory. *offset* is the offset of the unit wrt. the enclosing segment.

```
$Integer ↦ Integer;        λ SIZE
MemUnit ↦ (iTop, sTop, offset, size : Integer);
            λ UNIT, UNIT1, UNIT2, UNIT3, U
---
size(U):Integer ↦ U.size
offset(U):Integer ↦ U.offset
address(U):Integer ↦ U.iTop
---
memory → segment consisting of UNIT ↦
    UNIT.iTop = 0
    UNIT.offset = 0
UNIT → segment consisting of UNIT1 ↦
    UNIT1.iTop = UNIT.iTop
    UNIT.size = UNIT1.size
    UNIT1.offset = 0
    UNIT.sTop = UNIT1.sTop
UNIT → elementary of size SIZE ↦
    UNIT.sTop = UNIT.iTop + SIZE
    UNIT.size = SIZE
UNIT1 → UNIT2 concatenated UNIT3 ↦
    UNIT2.offset = UNIT1.offset
    UNIT3.offset = UNIT1.offset + UNIT2.size
    UNIT2.iTop = UNIT1.iTop
    UNIT3.iTop = UNIT2.sTop
    UNIT1.sTop = UNIT3.sTop
    UNIT1.size = UNIT2.size + UNIT3.size
UNIT1 → UNIT2 overlapped UNIT3 ↦
    UNIT2.offset = UNIT1.offset
    UNIT3.offset = UNIT1.offset
    UNIT2.iTop = UNIT1.iTop
    UNIT3.iTop = UNIT1.iTop
    UNIT1.sTop = max(UNIT2.sTop, UNIT3.sTop)
    UNIT1.size = max(UNIT2.size, UNIT3.size)
```

Fig. 3. Module "*Alloc*": Signature and Implementation.

3.3. Code generation

We define a code generator for the control structure of while-programs. The target language is represented by the relation symbols of the data type whose signature is listed in Fig. 4. As, with respect to code generation, the target language is a purely syntactic concept, we have chosen relation symbols to denote its basic constructs. The abstract target machine is assumed to have an unbounded number of registers *register*(i). The intuitive meaning of the constructs is as usual. *JifF* means jump, if false, on the first argument to the label that is given by the second argument.

$$
\begin{array}{l}
\Sigma Integer,\ Texp,\ Tstat\\
\text{---}\\
\text{---}\\
Texp \rightarrow \mathrm{const}(Integer)\\
Texp \rightarrow \mathrm{register}(Integer)\\
Texp \rightarrow \mathrm{cont}(Texp)\\
Texp \rightarrow Texp\ op\ Texp,\ op = +,-,*,/,\langle,\rangle,=,\ldots\\
Texp \rightarrow op(Texp)\ ,\ op = -,\ \mathrm{cont},\ \mathrm{not},\ldots\\
Tstat \rightarrow \mathrm{assign}(Texp,\ Texp)\\
Tstat \rightarrow \mathrm{JifF}(Texp,\ Texp)\\
Tstat \rightarrow \mathrm{Jmp}(Texp)\\
Tstat \rightarrow \mathrm{label}(Integer)\\
Tstat \rightarrow \mathrm{skip}\\
Tstat \rightarrow Tstat;\ Tstat
\end{array}
$$

11010

Fig. 4. Signature of target language.

Thus, the formula

$$
\begin{array}{l}
S \rightarrow S1;S1' \wedge S1' \rightarrow S2;S3\\
\quad \wedge S1 \rightarrow \mathrm{label}(1) \wedge S2 \rightarrow \mathrm{assign}(V, E) \wedge V \rightarrow V1 + V2 \wedge V1 \rightarrow \mathrm{register}(1)\\
\quad \wedge V2 \rightarrow \mathrm{const}(6) \wedge E \rightarrow E1 + E2 \wedge E1 \rightarrow \mathrm{cont}(E3) \wedge E3 \rightarrow E4 + E5\\
\quad \wedge E4 \rightarrow \mathrm{register}(1) \wedge E5 \rightarrow \mathrm{const}(5) \wedge E2 \rightarrow \mathrm{const}(1)\\
\quad \wedge S3 \rightarrow \mathrm{Jmp}(\mathrm{EL})
\end{array}
$$

would characterize a statement scheme $S \in Tstat$ containing a label variable EL over *Texp*. In what follows we will choose a more concise notation, if the names of the pairwise distinct auxiliary variables are not of interest. We write instead

$$
\begin{array}{l}
S \rightarrow \mathrm{label}(1);\\
\quad \mathrm{assign}(\mathrm{register}(1)+\mathrm{const}(6),\mathrm{cont}(\mathrm{register}(1)+\mathrm{const}(5))+\mathrm{const}(1));\\
\quad \mathrm{Jmp}(\mathrm{EL})
\end{array}
$$

i.e. view the relation symbols as context-free productions and identify sentential forms of this grammar with their parses. The parses themselves can be regarded as a particular kind of relational expressions where auxiliary variables (e.g., S1, S1′, ...) denote positions of nonterminals in intermediate sentential forms.

Figure 5 presents the source language related notions of imperative commands which the module *CodeGeneration* offers, i.e. a definition of a module code generation with representation

$$\rho_C : \Sigma_{CodeGeneration} \rightarrow \Sigma_{TargetLang}.$$

For implementing while-programs in terms of goto-programs, label counters *LabCtr* have to be maintained in order to generate unique numbers for label declarations. *c* is the target code to be generated. $\mathrm{V} \rightarrow \mathrm{mkVar}(\mathrm{B}, \mathrm{I})$ defines the ‘source language’

```
$Integer ↦ Integer;      λI, B
Svar ↦ Texp;   λV, V1, SVAR
Sexp ↦ Texp;   λE, E0, E1, E2, SEXP
Sstat ↦ (iLabCtr, sLabCtr: Integer, c: Tstat); λs, s0, s1, s2
---
---
V → mkVar(B, I) ↦ V → register(B) + const(I)
V1 → V offset I ↦ V1 → V + I
V1 → V[E] ↦ V1 → cont(V) + E
E → const(I) ↦ E → const(I)
E → mkExp(V) ↦ E → cont(V)
E0 → E1 op E2 ↦ E0 → E1 op E2
E → −E1 ↦ E → −E1
targetCode → s ↦
    s.iLabCtr = 0
s → skip ↦
    s.sLabCtr = s.iLabCtr
    s.c → skip
s0 → s1;s2 ↦
    s1.iLabCtr = s0.iLabCtr
    s2.iLabCtr = s1.sLabCtr
    s0.sLabCtr = s2.sLabCtr
    s0.c → s1.c;s2.c
s → SVAR := SEXP ↦
    s.sLabCtr = s.iLabCtr
    s.c → assign(SVAR, SEXP)
s0 → while SEXP do s1 ↦
    s0.c → label(s0.iLabCtr);
           JifF(SEXP, const(s0.iLabCtr + 1));
           s1.c;
           Jmp(const(s0.iLabCtr));
           label(s0.iLabCtr + 1)
    s1.iLabCtr = s0.iLabCtr + 2
    s0.sLabCtr = s1.sLabCtr
s0 → if SEXP then s1 else s2 ↦
    s0.c → JifF(SEXP, const(s0.iLabCtr));
           s1.c;
           Jmp(const(s0.iLabCtr + 1));
           label(s0.iLabCtr);
           s2.c;
           label(s0.iLabCtr + 1)
    s1.iLabCtr = s0.iLabCtr + 2
    s2.iLabCtr = s1.sLabCtr
    s0.sLabCtr = s2.sLabCtr
```

Fig. 5. Module "*Code Generation*": Signature and implementation.

construct *mkVar* in terms of the 'target language' constructs *register* and *const.* In v1 → v[i], v is the address of the dope vector of the array. Its first element contains the address of the array to which the index has to be added to yield the array element v1. The code templates (rules for calculating the *c*-components) are as usual. Except for standard types, the module *CodeGeneration* has no parameters.

3.4. Remarks

Properties of the imported operators and relations determine the properties of the newly defined relations. We can assume that e.g.

$$lookup(enter(enterScope(init), \text{“}x\text{”}, \text{“}integer\text{”}), \text{“}x\text{”}) = \text{“}integer\text{”}.$$

This implies that the formula

program → binding constructs C1 ∧
Cl → scope with binding constructs C2 ∧
C2 → C3 ; C4 ∧
C3 → bind "*x*" to "*integer*" ∧
$C4 \to C5 ; C6 \wedge$
$T = find(C5, \text{“}x\text{”})$

can be satisfied for $T = integer$, only. Whereas the definition of the relations refers to a specific *implementation* in terms of a given symbol table handling module, the *properties* of the relations can be expressed without referring to this implementation. They do directly model some language concept of binding identifiers to declarations in the presence of named and nested scopes. On the next higher level of the compiler specification only these properties are relevant. So much just for recalling the basic idea behind encapsulation of implementation (also called abstraction).

4. Combining modules to make compilers

According to [6] and [23], signature morphisms are the only syntactic mechanism needed for structuring data types. Semantically there are two aspects of signature morphisms σ: the forgetful functor σ-*struct*, cf. Section 2.2, and σ-persistent type generators

$$T : \Sigma\text{-}struct \to \Sigma'\text{-}struct,$$

cf. below. Combining data types means, therefore, applying a type generator or a forgetful functor.

Our application to compilers has required to define a version of signatures and signature morphisms that, in contrast to the standard approach, also includes relation symbols. Moreover, our signature morphisms in general map sorts to sequences of sorts, operators to terms, and relation symbols to relation expressions. So it needs to be demonstrated that this version of signature morphisms satisfies some basic requirements, allowing to adopt the structuring principles of abstract data type theory. In the following we will briefly state that these requirements are, in fact, satisfied.

In the formal presentation we follow Lipeck [23]. The proofs of the theorems given below are straightforward extensions to signature morphisms in our sense of Lipeck's proofs. The reader is assumed to be familiar with the basic notions and techniques of parameterized data types although the presentation below will be self-contained.

4.1. Parameterized data types

A (class of) *data type(s)* is a pair $D=(\Sigma, C)$, consisting of a signature Σ and a full sub-category $C \subseteq \Sigma\text{-}struct$ of Σ-structures that is closed under isomorphism.

A parameterized data type is a triple $P=(D, D1, T)$, where D and $D1$ are classes of data types such that

- $\Sigma 1 = \Sigma + (S1, \Omega 1, R1)$ and $i\text{-}struct(C1) \subseteq C$, if i is the inclusion morphism $\Sigma \subseteq \Sigma 1$,
- $T: C \to C1$ is an object-surjective functor.

Σ is the parameter signature, C the class of parameter structures. $\Sigma 1$ is the body signature and $C1$ the class of structures that is the range of the type constructor T. T is called *object-surjective*, if for any $\Sigma 1$-structure $c1 \in C1$ there exists a Σ-structure $c \in C$ such that $T(c) = c1$. P is called *persistent*, if $i\text{-}struct\ T = id_C$.

Example 4.1. We define a parameterized data type "*SymbolTable*" in a way such that it is a useful basis of the compiler module *Identification*. Its signature has been given in Section 3, cf. Fig. 1.

- *Parameter*:

C: set of all $A \in \Sigma\text{-}struct$ (Σ = parameter signature of *SymbolTable*) such that
- $Integer_A = Int$, the set of integers,
- $=, +, \ldots$ are the usual standard functions on Int,
- $Bool_A = \{true, false\}$,
- $DeclInfo_A$ contains some distinguished element $undefined_A$,[8]
- eq_A is an equivalence relation on Id_A.

- *Type constructor*: For $A \in C$ set $B = T(A)$, where

$$StStates_B = \\ maxsc{:}\mathbb{N}_0 \times st{:}\mathbb{N}_0^* \times dt{:}(s{:}\mathbb{N}_0 \times i{:}Id_A \times d{:}DeclInfo_A)^*$$

maxsc is the maximal scope number used so far. *st* is the scope table and *dt* is the declarations table. Both are lists. *st* is a list of (natural) scope numbers whereas

[8] In order to keep our examples small, we are not interested in providing any realistic error handling in the case of undeclared or multiply declared identifiers. In the former case the value $undefined_A$ will be returned. Therefore we need not include a specific constant undefined among the parameter constants in Σ.

dt is the list of declarations encountered so far. s, i, d are the scope, identifier, and declaration information, resp., of the declaration.

$init_B = (0, (0), (\ ))$

$enterScope_B(state) =$
$(state.maxsc + 1,\ state.st \text{ conc } state.maxsc + 1,\ state.dt)$

$leaveScope_B(state) =$
$(state.maxsc,\ \text{deleteLastElem}(state.st),\ state.dt)$

$currentScope_B(state) = \text{lastElem}(state.st)$

$enter_B(state, x, t) =$
$(state.maxsc,\ state.st,\ state.dt \text{ conc } (\text{lastElem}(state.st),\ x,\ t))$

The new declaration is appended to the existing list of declarations. In the case of redeclaration, dt now contains more than one entry for x.

$lookup_B(State, x) = t,$

if there exists $1 \leq l \leq \text{length}(state.st)$ and $1 \leq j \leq \text{length}(state.dt)$ such that $t = (state.dt)[j].d$, $eq(x, (state.dt)[j].i) = true$, and $state.st[l] = (state.dt[j]).s$. If more than one such l and/or j exist, then take first the maximal l and, then, the maximal j. If no such j and l exist, set $t = undefined_A$.

$lookupQual_B(state, b, x) = t,$

where t as above, if, additionally, l is chosen such that $state.st[l] = b$.

Note that we do not refer to any mechanism for specifying basic parameterized data types in the sense of abstract data type theory, i.e. we are not interested in how the classes C of parameter structures and the type constructors T are defined. In the above example we have used a semi-formal mathematical notation. In practical applications of the concept, data types that do not contain any relation symbols can be given as packages in some adequate imperative programming language, e.g. PASCAL. This makes the treatment independent of the properties of a specification language. In particular we need not be concerned about generating executable programs from their specifications. Here we are only interested in the mechanisms for combining complex data types out of given elementary ones. Once the basic data types are given in an efficiently executable form, the combining mechanisms to be introduced below will allow to obtain an efficient implementation of the combined data type automatically. In the following sections, the basic combinators for data types are introduced.

4.2. Parameter passing

Given parameterized types P and P' and given a signature morphism $\alpha : \Sigma \to \Sigma 1'$, P' is called an (admissible) actual parameter for P with respect to α, if $\alpha\text{-}struct(C1') \subseteq C$. α is called an actual *parameter association*.

Passing an actual parameter to a given parameterized type has a syntactic (resulting signature) and a semantic aspect (resulting type constructor). The result signature is modelled by pushouts. We, first, repeat the definition of pushouts.

A diagram (in SIG)

$$\begin{tikzcd} \Sigma \arrow[r, "\sigma 1"] \arrow[d, "\sigma 2"'] & \Sigma 1 \arrow[d, "\sigma' 2"] \\ \Sigma 2 \arrow[r, "\sigma' 1"] & \Sigma_{po} \end{tikzcd}$$

is called a pushout diagram, if

(1) it is a diagram, i.e. it commutes,

(2) for any signature Σ' and any pair of morphisms $f1: \Sigma 2 \to \Sigma'$ and $f2: \Sigma 1 \to \Sigma'$ such that $f1\sigma 2 = f2\sigma 1$, there exists a unique morphism $\sigma': \Sigma_{po} \to \Sigma'$ such that $\sigma'\sigma' i = fi$, $i = 1,2$.

In contrast to signature morphisms in the classical sense, where all pushouts exist, the following is true:

Theorem 4.1. *The category* SIG *of signature morphisms does not have all pushouts.*

Proof. The following is a counter example: Let $S = \{s\}$, $S1 = \{t1, t2\}$, $S2 = \{u1, u2\}$, $\Omega 1 = \Omega 2 = R1 = R2 = \emptyset$, $\sigma 1: s \mapsto t1t2$, and $\sigma 2: s \mapsto u1u2$. Then, for $\Sigma' = (\{v1, v2\}, \emptyset, \emptyset)$, $f_1: ui \mapsto vi$, and $f2: ti \mapsto vi$, $f2\sigma 1 = f1\sigma 2$. For a pushout Σ_{po}, there exists a $\sigma': \Sigma_{po} \to \Sigma'$ such that

$$\sigma'\sigma' 1(ui) = vi = \sigma'\sigma' 2(ti) \quad \text{and} \quad \sigma' 1(u1)\sigma' 1(u2) = \sigma' 2(t1)\sigma' 2(t2).$$

Thus, $\sigma' 1(u1) \geq \sigma' 2(t1)$ or $\sigma' 1(u2) \geq \sigma' 2(t2)$.[9] Wolog. we assume that the first alternative is true. Then, let us consider a signature $\Sigma'' = (\{x1, x2, x3\}, \emptyset, \emptyset)$ and morphisms $f''1: u1 \mapsto x1$, $u2 \mapsto x2x3$, $f''2: t1 \mapsto x1x2$, $t2 \mapsto x3$. This, again, leads to a commutative diagram $f''2\sigma 1 = f''1\sigma 2$. Thus, there has to exist a $\sigma'': \Sigma_{po} \to \Sigma''$ such that $\sigma''\sigma' i = f'' i$. Therefore,

$$f''2(t1) = x1x2 = \sigma''\sigma' 2(t1) \quad \text{and} \quad f''1(u1) = x1 = \sigma''\sigma' 1(u1).$$

In particular, $\sigma''(\sigma' 1(u1)) < \sigma''(\sigma' 2(t1))$, which contradicts to $\sigma' 2(t1) \leq \sigma' 1(u1)$. Consequently, no pushout can exist in this situation. □

The above proof shows that the introduction of relation symbols is not the reason for nonexisting pushouts. The crucial point is that morphism are allowed to send sorts, operators, and relation symbols to sequences, terms, and expressions in target sorts, operators, and relation symbols, respectively. The following theorem gives a sufficient and complete criterion for the existence of the sort part of pushouts. For operators and relation symbols, analogous assertions can be proved.

[9] $x \geq y$ means here that y is a substring of x.

Theorem 4.2. *Assume the above diagram to be given. Furthermore, assume that*

$$\Omega 1 = \Omega 2 = R1 = R2 = \emptyset.$$

Then, the diagram is a pushout diagram, iff S^*_{po} *(as a monoid) is isomorphic to* $(S1+S2)^*/\equiv$, *where* $\equiv$ *is the (least) congruence on* $(S1+S2)^*$ *generated by*

$$\{\sigma 1(s) \equiv \sigma 2(s) \mid s \in S\}.$$

Example 4.2. We apply the theorem to the counter example given in the proof of the last theorem. Let $S=\{s\}$, $S1=\{t1, t2\}$, $S2=\{u1, u2\}$, $\sigma 1: s \mapsto t1t2$, and $\sigma 2: s \mapsto u1u2$. Then, $\{[t1],[t2],[u1],[u2]\}$ is a minimal generator system for $(S1+S2)^*/\equiv$. As $[t1t2]=[u1u2]$, $(S1+S2)^*/\equiv$ is not freely generated.

For our purposes it is of interest that pushouts do exist if one of the two morphisms is an inclusion.

Theorem 4.3. *Given* $\sigma i: \Sigma \to \Sigma i$, $i=1, 2$, *two signature morphisms. If* $\Sigma \subseteq \Sigma 1$ *and* $\sigma 1$ *is the inclusion morphism, then there exists a signature* Σ_{po} *and morphisms* $\sigma' 1: \Sigma 2 \to \Sigma_{\text{po}}$ *and* $\sigma' 2: \Sigma 1 \to \Sigma_{\text{po}}$ *such that*

$$\begin{tikzcd}
\Sigma \arrow[r, "\sigma 1"] \arrow[d, "\sigma 2"'] & \Sigma 1 \arrow[d, "\sigma' 2"] \\
\Sigma 2 \arrow[r, "\sigma' 1"] & \Sigma_{\text{po}}
\end{tikzcd}$$

is a pushout diagram.

Proof. Define:

$S_{\text{po}} = (S1 - S) + S2$
$\sigma' 2(s) =$ **if** $s \in S$ **then** $\sigma 2(s)$ **else** s
$\Omega_{\text{po}} = \Omega 2 + \Omega 3,$
  where $\Omega 3 = \{f.i: \sigma' 2(u) s_i \mid f: us \in \Omega 1 - \Omega,\ \sigma' 2(s) = s_1 \ldots s_n,\ 1 \leqslant i \leqslant n\}$
$\sigma' 2(f) =$ **if** $f: us \in \Omega 1 - \Omega$, $\sigma' 2(s) = s_1 \ldots s_n$ **then** $(f.1, \ldots, f.n)$ **else** $\sigma 2(f)$
$R_{\text{po}} = R2 + R3$, where $R3 = \{r: \sigma' 2(u) \mid r: u \in R1 - R\}$
$\sigma' 2(r) =$ **if** $r \in R$ **then** $\sigma 2(r)$ **else** r
$\sigma' 1$ the inclusion $\Sigma 2 \subseteq \Sigma_{\text{po}}$.

It is straightforward to prove that with these definitions the above diagram is in fact a pushout diagram in SIG. The commutativity of the diagram follows immediately. The pushout property requires for any signature Σ' and any morphisms $f1: \Sigma 2 \to \Sigma'$, $f2: \Sigma 1 \to \Sigma'$ such that $f1 \sigma 2 = f2 \sigma 1$ the existence of a unique $\sigma': \Sigma_{\text{po}} \to \Sigma'$ such that

$\sigma'\sigma'1 = f1$ and $\sigma'\sigma'2 = f2$. σ' can be defined as follows:

$$\sigma'(s) = \textbf{if } s \in S2 \textbf{ then } f1(s) \textbf{ else } f2(s)$$
$$\sigma'(f) = f1(f), \quad \text{if } f{:}us \in \Omega 2.$$

Any operator $f.i{:}us$ in $\Omega 3$ has been obtained from a uniquely determined operator $f \in \Omega 1 - \Omega$ for which $\sigma'2(f) = (f.1, \ldots, f.k)$. It is not difficult to see that in this case

$$f2(f) = (g_{11}, \ldots, g_{1m_1}, g_{21}, \ldots, g_{2m_2}, \ldots, g_{k1}, \ldots, g_{km_k})$$

is a consequence of $f1\sigma 2 = f2\sigma 1$. With this,

$$\sigma'(f.i) = (g_{i1}, \ldots, g_{ik_i}).$$

Finally, for relation symbols $r \in \Omega_{\mathrm{po}}$,

$$\sigma'(r) = \textbf{if } r \in R2 \textbf{ then } f1(r) \textbf{ else } f2(r).$$

The properties $\sigma'\sigma'1 = f1$ and $\sigma'\sigma'2 = f2$ as well as the uniqueness of σ' follow immediately from the definition of σ'. $\square$

We have intentionally given the details of the definition of σ' as it is important for the implementation of the concept. Below we will give a theorem about how combinations of data types can be transformed into a normal form. The normal form will be the basis for compiler generation. The construction of σ' will be essential in the construction of the normal form.

Theorem 4.4 ([23]). *Let be given a pushout diagram as above. Furthermore, let K be an arbitrary category. Then, to any pair of functors $Ti{:}K \to \Sigma_i$-struct for which*

$$\sigma 1\text{-}struct\ T1 = \sigma 2\text{-}struct\ T2$$

there exists exactly one functor $T1 \cup T2{:}K \to \Sigma_{\mathrm{po}}$-struct for which

$$\sigma'1\text{-}struct\ (T1 \cup T2) = T2 \quad \text{and} \quad \sigma'2\text{-}struct\ (T1 \cup T2) = T1.$$

Proof. We give a short outline of the proof.

Define, for objects $k \in K$, $B = (T1 \cup T2)(k)$ as follows:

$$\text{for } x \in S2 \cup \Omega 2 \cup R2{:} x_B = x_{T2(k)},$$
$$\text{for } x \in (S1 - S2) \qquad {:} x_B = x_{T1(k)}.$$

This implies that for any $u \in S1^+$, $u_{T1(k)} = \sigma'2(u)_B$. (Indeed, for $1 \leq i \leq |u|$ we have, if $u.i \in S$,

$$u.i_{T1(k)} = u.i_{\sigma 1\text{-}struct(T1(k))} = u.i_{\sigma 2\text{-}struct(T2(k))} = \sigma 2(u.i)_{T2(k)} = \sigma'2(u.i)_{T2(k)}$$
$$= \sigma'2(u.i)_B.$$

If $u.i \in S1 - S$, from the definition of B, $u.i_B = u.i_{T1(k)}$.) Then, we further define, for $r \in R1 - R$, $r_B = r_{T1(k)}$. This is well-defined as $r_{T1(k)} \subseteq u_{T1(k)} = \sigma'2(u)_B$ and $\sigma'2(r)$ is of sort $\sigma'2(u)$. Finally, for $f.i \in \Omega 3$, it is $f{:}us \in \Omega 1$ and $f.i = \sigma'2(f).i{:}\sigma'2(u) \to$

$\sigma'2(s).i$. Thus, $f_{T1(k)}: u_{T1(k)} \to s_{T1(k)}$, i.e. $f_{T1(k)}: \sigma'2(u)_B \to \sigma'2(s)_B$. If $p_i : \sigma'2(s)_B \to (\sigma'2(s).i)_B$ is the projection to the ith component, set

$$f.i_B = p_i f_{T1(k)} : \sigma'2(u)_B \to (\sigma'2(s).i)_B.$$

For morphisms h in K define $g = (T1 \cup T2)(h)$ as follows:

$$g_s = \textbf{if } s \in S \textbf{ then } T2(h)_{\sigma 2(s)} \textbf{ else } T1(h)_s.$$

Considerations similar to the above show that g is in fact a morphism and that $T1 \cup T2$ is a functor. The uniqueness of $T1 \cup T2$ follows immediately from the construction. □

Parameter passing is now defined as follows. If P_2 is an admissible parameter for P_1 wrt. α (both persistent parameterized types), then consider the pushout diagram where $\sigma 1$ is the inclusion $\Sigma \subseteq \Sigma_1$ and where $\sigma 2 = \alpha$. Then, the result of applying P_2 to P_1 according to α is given as

$$\textbf{apply}\ (P_1, P_2, \alpha) = (D_2, D, T),$$

where $D = (\Sigma_{po}, T(C2))$, $T = T'_1 T_2$, if $T'_1 = (id_{C1_2} \cup (T_1 \alpha\text{-}struct))$. This situation will in the following be denoted by the diagram

$$\begin{tikzcd}
& D_1 \arrow[r, "\sigma_1;\, T_1"] \arrow[d, "\alpha"] & D1_1 \arrow[d, "\alpha'"] \\
D_2 \arrow[r, "\sigma_2;\, T_2"] & D1_2 \arrow[r, "\sigma'_1;\, T'_1"] & D
\end{tikzcd}$$

4.3. *Abstraction*

The second basic operation on parameterized data types is called *abstraction* (or reduction). Abstraction models in our applications a kind of implementation of a data type of signature Σ_2 over a data type of signature Σ_1.

Given P, $\Sigma \subseteq \Sigma 2$, and arbitrary $\sigma : \Sigma 2 \to \Sigma 1$ such that $\sigma|_{\Sigma} = id_{\Sigma}$,[10] then

$$\textbf{abstract}(P, \sigma) = (D, D2, T2),$$

where $D2 = (\Sigma 2, \sigma\text{-}struct(C1))$, $T2 = \sigma\text{-}struct\ T$.

Combinations of data types are terms in **apply** and **abstract** over basic data types and signature morphism. Most terms can mechanically be reduced to terms in which **abstract** occurs exactly once, namely at the root of the term. This is the assertion of the following normal form theorem that was proved in [23] and which can be adapted to our case.

[10] It would be sufficient to require that $\sigma'\sigma^{-1}$, with σ' the inclusion $\Sigma \subseteq \Sigma 1$, is an injective signature morphism.

Theorem 4.5. *If* **apply(abstract(**P_1, ρ_1**), abstract(**P_2, ρ_2**),** α**)** *is defined, then there exists a signature morphism* ρ *such that this term is equal to*

abstract(apply($P_1, P_2, \rho_2\alpha$**),** ρ**).**

Proof. We repeat the proof of [23] as we are interested in the construction of ρ. The diagram below shows the general situation:

$$\begin{tikzcd}
 & D_1 \arrow[rr, "\bar\sigma_1;\bar T_1"] \arrow[drrr, "\sigma_1;T_1"] \arrow[dd, "\alpha"] & & \bullet \arrow[dr, "\rho_1"] \arrow[dd, "\alpha'"] & \\
 & & & & D1_1 \arrow[dd, "(\rho_2\alpha)'"] \\
D_2 \arrow[r, "\bar\sigma_2;\bar T_2"] \arrow[drr, "\sigma_2;T_2"'] & \bullet \arrow[rr, "\bar\sigma'_1;\bar T'_1"] \arrow[dr, "\rho_2"] & & D' \arrow[dr, "\rho"] & \\
 & & D1_2 \arrow[rr, "\sigma'_1;T'_1"'] & & D
\end{tikzcd}$$

Here, the pushout $(\alpha, \bar\sigma_1)$ is the result of parameter passing in

apply(abstract(P_1, ρ_1**), abstract(**P_2, ρ_2**),** α**)**

whereas the pushout $(\rho_2\alpha, \sigma_1)$ represents

apply($P_1, P_2, \rho_2\alpha$**).**

This yields $\sigma'_1\rho_2\alpha = (\rho_2\alpha)'\rho_1\bar\sigma_1$. The pushout property of $(\alpha, \bar\sigma_1)$ implies the existence of a data type morphism $\rho: D' \to D$ such that $\rho\bar\sigma'_1 = \sigma'_1\rho_2$ and $\rho\alpha' = (\rho_2\alpha)'\rho_1$. This ρ makes the application of **abstract** in the second term to be well-defined as $\sigma'_1\sigma_2 = \rho\bar\sigma'_1\bar\sigma_2$. It remains to be shown that **abstract(**P, ρ**)** $= P'$, for $P = (D_2, D, T'_1 T_2)$ and $P' = (D_2, D', \bar T'_1 \bar T_2)$. The syntactic part of this equation has already been proved. To show ρ-*struct* $T'_1 T_2 = \bar T'_1 \bar T_2$ it needs, because of the last theorem, to be proved that

$$\alpha\rho\text{-}struct\ T'_1 T_2 = \bar\sigma'_1 \bar T'_1 \bar T_2.$$

This, however, is straightforward from the construction. □

This normal form theorem is the basis for an implementation of the concept in a compiler generating system:

(1) Predefined data types such as *SymbolTable* or *TargetLang* can be assumed to be given as a concrete package in a suitable language, say PASCAL.

(2) The compiler definition is a term in predefined data types, signature morphisms, **apply,** and **abstract.** The compiler generator must allow for reading in and storing signature morphisms in the general form.

(3) The compiler generator applies the algebraic laws for **apply** and **abstract.** This reduces the compiler definition to a term of kind

$$Comp = \mathbf{abstract}(Predefined, \rho)$$

where *Predefined* is an **apply**-term in predefined data types only. The latter corresponds to a number of textual expansions of the PASCAL-text of the predefined packages.

(4) ρ is checked as to whether it is an attribute grammar. If this is the case, it is transmitted to that part of the system that generates attribute evaluators.

Of course, the question is how it can be guaranteed that the transformed version of the compiler definition ρ is in fact an attribute grammar. This question will be the subject of the Section 5.

4.4. *Example: Compiler for a language L1*

4.4.1. *Combining the modules*

For the compiler modules of the last sections it holds

$$\begin{aligned}&Identification = \mathbf{abstract}(SymbolTable, \rho_I),\\ &Alloc = \mathbf{abstract}(Standard, \rho_A),\\ &CodeGeneration = \mathbf{abstract}(TargetLang, \rho_C).\end{aligned}$$

We now combine these data types into a compiler for a simple language L1. L1 is a language of while-programs with a block concept and with integer variables only. Now, the syntactic rules of the concrete language are the relation symbols whose implementation in terms of given data types is to be defined.

First, we combine all three modules by **apply:**

$$\begin{aligned}&Alloc + CodeGen = \mathbf{apply}(Alloc, CodeGeneration, id_{Standard}),^{11}\\ &AllModules = \mathbf{apply}(Alloc + CodeGen, Identification, id_{Standard}),\\ &L1Modules = \mathbf{apply}(AllModules, Standard, \{Id_{Standard}, Id \to Id, eq \to eq,\\ &\qquad\qquad DeclInfo \to Integer\}).\end{aligned}$$

AllModules is the sum of all three modules. *L1Modules* is the result of passing *Integer* to *DeclInfo.* In this L1-compiler, addresses are the only relevant information about (variable) identifiers.

[11] The standard types are in this case the only parameters of *Alloc.* $id_{Standard}$ is the identity signature morphism on standard types. In this application **apply** means disjoint union of data types without duplicating standard types.

Figures 6 and 7 depict

$$\rho_{L1}: \Sigma_{L1} \to \Sigma_{L1Modules}.$$

Then, the L1-compiler is given as the data type

$$L1Comp = \textbf{abstract}(L1Modules, \rho_{L1}).$$

Block, Statement ↦ (*env*: *Bindings*, *memory*: *MemUnit*, *code*: *Sstat*)
Decl ↦ (*env*: *Bindings*, *memory*: *MemUnit*)
$*Id* ↦ *Id*
Var ↦ (*env*: *Bindings*, *code*: *Svar*)
Exp, Bexp ↦ (*env*: *Bindings*, *code*: *Sexp*)

$*eq*(*Id*, *Id*): *Bool* ↦ *eq*(*Id*, *Id*): *Bool*
---
prog → BLOCK ↦
    program → binding constructs BLOCK.*env*
    memory → segment consisting of BLOCK.*memory*
    targetCode → BLOCK.*code*
BLOCK → **begin** DECL ; STAT **end** ↦
    BLOCK.*env* → scope with binding constructs *C*
    C → DECL.*env* ; STAT.*env*
    BLOCK.*memory* → segment consisting of *U*
    U → DECL.*memory* concatenated STAT.*memory*
    BLOCK.*code* = STAT.*code*
DECL0 → DECL1 ; DECL2 ↦
    DECL0.*env* → DECL1.env ; DECL2.*env*
    DECL0.*memory* → DECL1.*memory* concatenated DECL2.*memory*
DECL → ID : **integer** ↦
    DECL.*env* → bind ID to *address* (DECL.*memory*)
    DECL.*memory* → elementary of size 1
STAT → BLOCK ↦
    STAT.*env* = BLOCK.*env*
    STAT.*memory* = BLOCK.*memory*
    STAT.*code* = BLOCK.*code*

Fig. 6. Definition of the L1-compiler (Part 1).

One might visualize the definition of the L1-compiler as consisting of three partial translations that classify any L1-construct with respect to the abstract constructs as they are 'input' to the *Identification*, *Alloc*, and *CodeGeneration* module, respectively. The three translations are not independent of each other. E.g., in the definition of DECL → ID:**integer**, the modules *Identification* and *Alloc* interact with each other: *Alloc* determines the address of the variable which is then used to describe the kind of the identifier in its declaration. Integer variables are assumed to occupy storage

STAT0 → STAT1 ; STAT2 ↦
    STAT0.*env* → STAT1.*env* ; STAT2.*env*
    STAT0.*memory* → STAT1.*memory* overlapped STAT2.*memory*
    STAT0.*code* → STAT1.*code* ; STAT2.*code*
STAT → V := EXP ↦
    STAT.*memory* → elementary of size 0
    STAT.*env* → V.*env*;EXP.*env*
    STAT.*code* → V.*code* := EXP.*code*
V → ID ↦
    V.*env* → noBindings
    V.*code* → mkVar(1, *find*(V.*env*, ID))
EXP → I ↦
    EXP.*env* → noBindings
    EXP.*code* → const(I)
EXP → V ↦
    EXP.*env* = V.*env*
    EXP.*code* → mkExp(V.*code*)
EXP0 → EXP1 *op* EXP2 (*op* = +, −, *, /) ↦
    EXP0.*env* → EXP1.*env*;EXP2.*env*
    EXP0.*code* → EXP1.*code* *op* EXP2.*code*
STAT0 → **while** BEXP **do** STAT1 ↦
    STAT0.*env* → BEXP.*env* ; STAT1.*env*
    STAT0.*memory* = STAT1.*memory*
    STAT0.*code* → **while** BEXP.*code* **do** STAT1.*code*
BEXP → EXP1 *relop* EXP2 (*relop* = ⟨, ⟩, =, . . .) ↦
    BEXP.*env* → EXP1.*env* ; EXP2.*env*
    BEXP.*code* → EXP1.*code* *relop* EXP2.*code*
STAT0 → **if** BEXP **then** STAT1 ↦
    STAT0.*env* → BEXP.*env* ; STAT1.*env*
    STAT0.*memory* = STAT1.*memory*
    STAT0.*code* → **if** BEXP.*code* **then** STAT1.*code* **else** *S*
    S → *skip*

Fig. 7. Definition of the L1-compiler (continued).

of size 1 (say, word). Concerning the addressing of program variables it is assumed that Register 1 holds the address of the program memory. *find*(V.*env*, ID) is, then, the offset of the variable in the activation record.

Just to remind the reader of the (meta-) semantics of the definition of ρ_{L1}, let us consider the implementation of the block relation. It holds

$(B, D, S) \in$ "_ → *begin*_;_*end*" $\Leftrightarrow$
$(B.env, C) \in$ "_ → scope with binding constructs _" $\wedge$
  $(C, D.env, S.env) \in$ "_ → _ ; _" $\wedge$
  $(B.memory, U) \in$ "_ → segment consisting of _" $\wedge$
  $(U, D.memory, S.memory) \in$ _ → _concatenated _" $\wedge$
  $B.code = S.code.$

4.4.2. The expanded version of the L1-compiler

According to the normal form theorem, it holds

$$L1\,Comp = \mathbf{abstract}(\mathbf{abstract}(Predefined, \rho), \rho_{L1}) \\ = \mathbf{abstract}(Predefined, \rho\rho_{L1}), \quad \text{for some } \rho,$$

where *Predefined* is an **apply** term that unites the elementary data types *Standard*, *TargetLanguage*, and *SymbolTable* and passes *Integer* to *DeclInfo*. Figure 8 shows part of $\rho'_{L1} = \rho\rho_{L1}$ as it is implicit in the proof of the normal form theorem. In this example, ρ'_{L1} is again an attribute grammar although ρ_{L1} is not, i.e. the definition of *L1 Comp* could be subject to automatic compiler generation. Appendix 1 lists the complete definition of ρ' of the L1-compiler.

The reader should realize that descriptions such as ρ'_{L1} are input to compiler generating systems today. They are quite unstructured. In particular, they do not exhibit any language concept that the compiler has to cope with. This may be acceptable for languages as simple as L1. If the language, however, gets more complex, then structuring is a must. In such cases, attribute grammars can be very big. A notable example is the definition of the Karlsruhe ADA compiler [1]. This document presents a 20 000 lines attribute grammar specifying the static semantics of roughly 270 syntactic rules using about 60 different attributes. We believe that the structuring concepts introduced above could convert even such a big compiler definition into something readable and manageable.

In general it should be obvious that the possibility of deriving compiler descriptions modularly out of modules that correspond to the language facets increases flexibility and modifiability considerably. Note also that the modules which we have defined above would allow to define compilers for much more realistic languages, too. We show this by defining the compiler for a second simple language L2.

4.5. Example: A compiler for the language L2

L2 is a language of while-programs with type declarations and record types. We show that L2 is just a different combination of the same concepts which L1 involved. It should be obvious that if the compilers for both L1 and L2 can be obtained as combinations of the previously defined modules, then this would also be possible for a language that is the sum of the concepts of both languages. The latter would be a fairly realistic language.

First, we construct a suitable combination of our basic modules. In L2, declaration information is more complex. To be able to instantiate the parameter *DeclInfo* of *SymbolTable* correspondingly, we assume a data type *L2Types* to be pregiven. Its signature is shown in Fig. 9. Its meaning is informally described as follows. The address a and the type t constitute the information $mkVarInfo(a, t)$ about a variable identifier. If the identifier names a field of a record, a is the offset of the field. Otherwise, it is the address of the variable. *Integer* is the only elementary type. The size (in memory units) s and the type t constitute the information $mkTypeIdInfo(s, t)$

```
. . .
Statements ↦
    ((env.iSt : StStates, env.sSt : StStates),
     (memory.iTop, memory.sTop, memory.offset, memory.size : Integer),
     (code.iLabCtr, code.sLabCtr : Integer, code.c : Tstat))
. . .
BLOCK → begin DECL ; STAT end ↦
    C.iSt = enterScope (BLOCK.env.iSt)
    BLOCK.env.sSt = leaveScope(C.sSt)
    DECL.env.iSt = C.iSt
    STAT.env.iSt = DECL.env.sSt
    C.sSt = STAT.env.sSt
    U.iTop = BLOCK.memory.iTop
    BLOCK.memory.size = U.size
    U.offset = 0
    BLOCK.memory.sTop = U.sTop
    DECL.memory.offset = U.offset
    STAT.memory.offset = U.offset + DECL.memory.size
    DECL.memory.iTop = U.iTop
    STAT.memory.iTop = DECL.memory.sTop
    U.sTop = STAT.memory.sTop
    U.size = DECL.memory.size + STAT.memory.size
    STAT.code.iLabCtr = BLOCK.code.iLabCtr
    BLOCK.code.sLabCtr = STAT.code.sLabCtr
    BLOCK.code.c = STAT.code.c
. . .
STAT0 → if BEXP then STAT1 ↦
    BEXP.env.iSt = STAT0.env.iSt
    STAT1.env.iSt = BEXP.env.sSt
    STAT0.env.sSt = STAT1.env.sSt
    STAT1.memory.offset = STAT0.memory.offset
    STAT1.memory.iTop = STAT0.memory.iTop
    STAT0.Memory.sTop = STAT1.memory.sTop
    STAT0.memory.size = STAT1.memory.size
    STAT0.code.c → JifF(BEXP.code,const(STAT0.code.iLabCtr));
        STAT1.code.c;
        Jmp(const(STAT0.code.iLabCtr + 1));
        label(STAT0.code.iLabCtr);
        S.c;
        label(STAT0.code.iLabCtr + 1)
    STAT1.code.iLabCtr = STAT0.code.iLabCtr + 2
    S.iLabCtr = STAT1.code.sLabCtr
    STAT0.code.sLabCtr = S.sLabCtr
    S.sLabCtr = S.iLabCtr
    S.c → skip
. . .
```

Fig. 8. L1-compiler: Version ρ'.

TypeDenotation, IdInfo, $Id

$eq(Id, Id) : Bool
mkVarInfo(Integer, TypeDenotation) : IdInfo
mkTypeIdInfo(Integer, TypeDenotation) : IdInfo
intType : TypeDenotation
recordType(Integer) : TypeDenotation
scopeNmb(TypeDenotation) : Integer
getType(IdInfo) : TypeDenotation
getAddress(IdInfo) : Integer
getSize(TypeDenotation) : Integer

Fig. 9. Data type *L2Types*: Signature.

about type identifiers. A record type *recordtype(s)* is viewed as the scope *s* in which the field names are defined. Scopes are numbered by integers. *scopeNmb* retrieves the number of the scope which the (record-) type defines.

We now define

$$L2Modules = \textbf{apply}(AllModules, L2Types, \{id_{Standard}, DeclInfo \to IdInfo, Id \to Id, eq \to eq\}).$$

Figures 10 and 11 give the definition of

$$\rho_{L2} : \Sigma_{L2} \to \Sigma_{L2Modules}$$

such that the L2-compiler is given as

$$L2Comp = \textbf{abstract}(L2Modules, \rho_{L2}).$$

Here record types constitute scopes of visibility for the names of the record fields. These names are retrieved using *findQual*(_,_,_). With respect to storage allocation, record types are segments so that the offset in these segments are associated with the field names. Declarations of type identifiers do not require any memory to be allocated.

A program variable can be a record variable. In this case the associated record number *recordNmb* identifies the record (scope) in which its field names are declared.

Note that the languages L1 and L2 are fairly different from each other. L1 has a block concept and primitive types only. L2 has no block concept but allows for record variables and for declarations of type identifiers. Moreover there are slight differences with respect to control constructs. Nevertheless, the compilers for both languages can be constructed as combinations of the same modules. This demonstrates to some extent the language independency of these modules.

Stat $\mapsto$ (*env* : *Bindings*, *code* : *Sstat*)
Decl, *Field* $\mapsto$ (*env* : *Bindings*, *memory* : *MemUnit*)
TypeDen $\mapsto$ (*env* : *Bindings*, *memory* : *MemUnit*, *type* : *TypeDenotation*)
\$*Id* $\mapsto$ *Id*
Var $\mapsto$ (*env* : *Bindings*, *code* : *Svar*, *recordNmb* : *Integer*)
Exp $\mapsto$ (*env* : *Bindings*, *code* : *Sexp*)

\$*eq*(*Id*, *Id*) : *Bool* $\mapsto$ *eq*(*Id*, *Id*) : *Bool*
---
program → **begin** DECL;STAT **end** $\mapsto$
  program → binding constructs *C*
  C → scope with binding constructs *C'*
  C' → DECL.*env* ; STAT.*env*
  memory → segment consisting of DECL.*memory*
  targetCode → STAT.*code*
DECL0 → DECL1 ; DECL2 $\mapsto$
  DECL0.*env* → DECL1.*env* ; DECL2.*env*
  DECL0.*memory* → DECL1.*memory* concatenated DECL2.*memory*
DECL → **type** ID = TYPEDEN $\mapsto$
  DECL.*env* → TYPEDEN.*env* ; *D*
  D → bind ID to *mkTypeIdInfo*(*size*(TYPEDEN.*memory*),TYPEDEN.*type*)
  DECL.*memory* → elementary of size 0
  memory → segment consisting of TYPEDEN.*memory*
DECL → **var** ID:TYPEDEN $\mapsto$
  DECL.*env* → TYPEDEN.*env* ; *D*
  D → bind ID to *mkVarInfo*(*address*(*decl.memory*),TYPEDEN.*type*)
  DECL.*memory* = TYPEDEN.*memory*
TYPEDEN → **integer** $\mapsto$
  TYPEDEN.*type* = *intType*
  TYPEDEN.*memory* → elementary of size 1
TYPEDEN → **record** FIELD **end** $\mapsto$
  TYPEDEN.*env* → scope with binding constructs FIELD.*env*
  TYPEDEN.*memory* → segment consisting of FIELD.*memory*
  TYPEDEN.*type* = *recordType*(*scope*(TYPEDEN.*env*))

Fig. 10. L2-compiler (part 1).

5. Implementation of the concept in a compiler-compiler

The examples have demonstrated that attribute grammars are a proper subclass of signature morphisms. We have also investigated how compiler definitions can modularly be composed out of elementary modules. Here, the modules as well as the final compiler is formally given as a signature morphism. Abstraction and parameter passing are the basic operations to compose compiler definitions.

```
TYPEDEN → ID ↦
    TYPEDEN.type = getType(find(TYPEDEN.env, ID))
    TYPEDEN.memory → elementary of size getSize(find(TYPEDEN.env, ID))
FIELD0 → FIELD1 ; FIELD2 ↦
    FIELD0.env → FIELD1.env ; FIELD2.env
    FIELD0.memory → FIELD1.memory concatenated FIELD2.memory
FIELD → ID : TYPEDEN ↦
    FIELD.env → TYPEDEN.env ; D
    D → bind ID to mkVarInfo(offset(FIELD.memory),TYPEDEN.type)
    FIELD.memory = TYPEDEN.memory
STAT0 → STAT1 ; STAT2 ↦
    STAT0.env → STAT1.env ; STAT2.env
    STAT0.memory → STAT1.memory concatenated STAT2.memory
    STAT0.code → STAT1.code ; STAT2.code
STAT → V := EXP ↦
    STAT.env → V.env;EXP.env
    STAT.memory → elementary of size 0
    STAT.code → V.code := EXP.code
V → ID ↦
    V.env → noBindings
    V.code → mkVar(1,getAddress(find(V.env,ID)))
    V.recordNmb = scopeNmb(getType(find(V.env,ID)))
V0 → V1.ID ↦
    V0.env = V1.env
    V0.code → V1.code offset getAddress(findQual(V0.env, V1.recordNmb,ID))
    V0.recordNmb = scopeNmb(getType(findQual(V0.env, V1.recordNmb,ID)))
EXP → I ↦
    EXP.env → noBindings
    EXP.code → const(I)
EXP → V ↦
    EXP.env = V.env
    EXP.code → mkExp(V.code)
EXP0 → EXP1 op EXP2; op = +, −, *, / ↦
    EXP0.env → EXP1.env ; EXP2.env
    EXP0.code → EXP1.code op EXP2.code
BEXP → EXP1 relop EXP2, relop = ⟨,⟩, =, . . . ↦
    BEXP.env → EXP1.env ; EXP2.env
    BEXP.code → EXP1.code relop EXP2.code
STAT0 → if BEXP then STAT1 else STAT2 ↦
    STAT0.env → BEXP.env ; E
    E → STAT1.env ; STAT2.env
    STAT0.code → if BEXP.code then STAT1.code else STAT2.code
```

Fig. 11. L2-compiler (continued).

To employ these methods in a compiler generating system based on attribute grammars, of interest are ways to guarantee that the resulting compiler description can in fact be considered as an attribute grammar rather than a more general signature morphism.

The situation would be unproblematic, if attribute grammars constituted a subclass of signature morphisms that has 'nice' properties. This, however, is not the case as demonstrated in the following section.

5.1. *Attribute grammars*

Attribute grammars are signature morphisms $\sigma : \Sigma \to \Sigma'$ where the (names of the) projections to the components of the $\sigma_S(s)$ are called the *attributes* of s,[12] where any relation symbol in Σ is written as $X_0 \to X_1 \cdots X_n$ and represents a *syntactic rule*, and where the relation expression $\text{AR} = \sigma(X_0 \to X_1 \cdots X_n)$ is the conjunction of *attribute evaluation rules* associated with $X_0 \to X_1 \cdots X_n$. Any atomic formula of AR must be of form

$$X_{i_0}.j_0 = f(X_{i_1}.j_1, \ldots, X_{i_k}.j_k),$$

i.e. AR do not make use of the relation symbols of Σ'. This asymmetry accounts for the fact that the composition of two attribute grammars is uninteresting: it is impossible to define a module as an attribute grammar and to call it from another attribute grammar. The 'output' of the caller is a term in $T_{\Omega'}$ and does therefore not connect to the 'input grammar' of relation symbols of the module. On the other hand, consider the module *Identification* of Section 3. Suppose, the module is composed with a signature morphism of the following kind:

$$\begin{aligned} & X \to YZ \mapsto \\ & \qquad X.env \to \text{scope with binding constructs } Y.env \\ & \qquad X.env' \to Y.env\,; Z.env \end{aligned}$$

The composition of the two morphisms results in

$$\begin{aligned} & X \to YZ \mapsto \\ & \qquad Y.env.iSt = enterScope(X.env.iSt) \\ & \qquad X.env.sSt = leaveScope(Y.env.sSt) \\ & \qquad Y.env.iSt = X.env'.iSt \\ & \qquad Z.env.iSt = Y.env.sSt \\ & \qquad X.env'.sSt = Z.env.sSt \end{aligned}$$

which is not an attribute grammar since two different rules for the attribute *env.iSt* at Y are now associated with one syntactic rule. This is not allowed for attribute grammars. Thus, it is necessary to generalize the notion of an attribute grammar.

[12] Remember that we have required that $\sigma_S(s)$ is nonempty so that grammar symbols without attributes are not allowed. This requirement could in principle be relaxed. Note, however, that a syntactic constructs 'without' semantics makes no sense.

At the same time this will correspond to the introduction of a restricted class of signature morphisms where the operators and relation symbols do more closely model semantic and syntactic properties, respectively.

Before introducing the formal notions, we illustrate the ideas. Let us consider the rule for declarations in L1:

DECL → ID:**integer** ↦
    DECL.*env* → bind ID to *address* (DECL.*memory*)
    DECL.*memory* → elementary of size 1

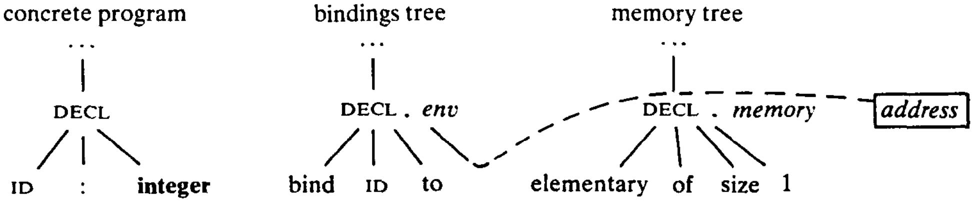
The corresponding rules in *Identification* and *Alloc* have been

B → bind ID to DECLINFO ↦
    B.*sSt* = enter (B.*iSt*;ID;DECLINFO)
UNIT → elementary of size SIZE ↦
    UNIT.*sTop* = UNIT.*iTop* + SIZE
    UNIT.size = SIZE

Let us for the moment visualize _→bind_to_ and _→elementary of size_ as uninterpreted functions

bind(*Id*, *DeclInfo*) : *Bindings*
elementary(*Integer*) : *MemUnit*.

Then, the rule for L1-declaration would become an ordinary attribute grammar rule. As the functions are uninterpreted, the standard interpretation applies, i.e. the values for the env and memory attributes would be terms in the operators *bind*, *elementary*, etc. These terms can then be considered as parse trees that are subject to attribute evaluation according to the attribute rules given in *Identification* and *Alloc*, respectively. The following figure illustrates the situation:



So the basic idea is to view relations in the semantic rules as tree templates that are themselves subject to attribute evaluation. Applying an operator. e.g. *address*, to a node in such a tree template corresponds to referring to an attribute at that node. In other words, the composition $\sigma'\sigma$ of two attribute grammars can be seen as consisting of two steps:

(1) Given a source program tree, attribute evaluation according to a relational rule means constructing a graph template and associating an output node of the graph as value with the attribute. It has to be guaranteed that this graph is a tree rather than a general acyclic graph.

(2) Perform attribute evaluation for the output trees of step (1) according to σ'.

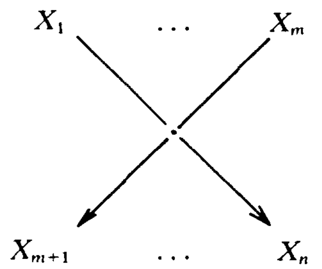
Thus, the problem is to guarantee that the graph of dependencies between attributes that are evaluated according to relational rules can in fact be viewed as a derivation graph according to some formal grammar.

5.2. *Language morphisms*

The basic idea will be as follows. Relation symbols denote facts about the syntactic environment of a construct in a program, whereas operators yield semantic information about a syntactic construct. Therefore, we require signatures $\Sigma = (S, \Omega, R)$ to additionally provide a partition of the sorts $S = SY + SM$ into *syntactic* sorts SY and *semantic* sorts SM. For any operator $f: s_1 \cdots s_n s_0 \in \Omega$ it is required that its result sort s_0 is a semantic sort in SM. Additionally, to any relation symbol $r: s_1 \cdots s_n \in R$ there shall exist a classification $\delta_r(i) \in \{in, out\}$ of any argument position $1 \leq i \leq n$ of r such that any *in*-position i is of a syntactic sort, i.e. $\delta_r(i) = in$ implies $s_i \in SY$. If

$$r: X_1 \cdots X_m X_{m+1} \cdots X_n$$

is a relation symbol with $\delta_r(i) = in$, for $1 \leq i \leq m$ and $\delta_r(i) = out$, for $m+1 \leq i \leq n$, the directions *in* and *out* refer to input nodes and output nodes resp., in a graph scheme of the following kind:



This situation is intuitively captured by our linear notation of relation symbols as grammar rules

$$r \equiv \tilde{\ } X_1 \tilde{\ } \cdots \tilde{\ } X_m \tilde{\ } \rightarrow \tilde{\ } X_{m+1} \tilde{\ } \cdots \tilde{\ } X_n \tilde{\ }$$

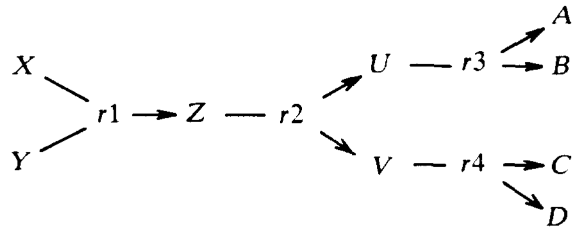
where $\tilde{\ }$ stands for the 'terminal' symbols in r.

From now on we will consider only signatures of this restricted kind. $\delta_r(i)$ is called the *direction* of argument i in r.

The conjunction of relations is now reflected by composing graph templates by identifying the input nodes of one template with the output nodes of the second template. E.g. the relation expression

$$E \equiv (XY \rightarrow r1Z \wedge Z \rightarrow r2UV \wedge U \rightarrow r3AB \wedge V \rightarrow r4CD)$$

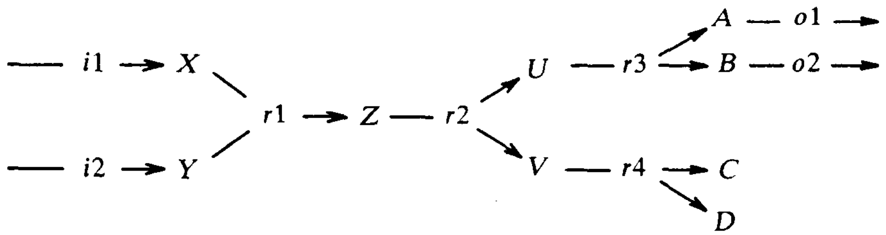
can graphically be written as



Here, X and Y are the *in*-nodes and $A \cdots D$ the *out*-nodes of the graph. This once more illustrates our view of relation expressions as graph constructors. The graph in the above example has the property that any node which is not an *in*-node has exactly one incoming edge and that any node which is not an *out*-node possesses exactly one outcoming edge. We call a graph of this kind *complete*, if it, additionally, does not have any *in*-node and if any *out*-node is of a semantic sort. E.g., the graph for

$$E' \equiv (E \wedge \rightarrow i1X \wedge \rightarrow i2Y \wedge A \rightarrow o1 \wedge B \rightarrow o2)$$

is given as



is complete, if C and D are of a semantic sort. Complete graphs can be viewed as graphs that represent the derivation of some terminal symbol from the empty string ε according to a Chomsky-0 type grammar. The remaining *out*-nodes have to be of semantic sort. They represent semantic parameters of the syntactic construct associated with a 'terminal' symbol in the syntactic rule. We will now characterize a subclass of signature morphisms that send relation expressions that are complete graphs in this sense again to complete graphs.

As indicated above, for sorts s the (names of the) projections to the components of $\sigma(s)$ are called attributes. Attributes in the classical sense are classified into inherited and synthesized attributes. The following definition provides for such a classification: We call a signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ *sort-classified*, if for any $s \in S$ there exists a classification $\delta_\sigma(s, i) \in \{in, out\}$ of any position $1 \leq i \leq n$ in $\sigma(s) = s_1 \cdots s_n$. Here, *in* stands for inherited and *out* for synthesized, respectively.

For sort-classified morphisms, we introduce the following notions:

- A *syntactic rule* p is a relation expression $r(x_1{:}s_1, \ldots, x_n{:}s_n)$ where $r{:}s_1 \ldots s_n \in R$ and where the $x_i{:}s_i$ are pairwise distinct variables.
- Given a syntactic rule p as above, an *attribute position* in p is a variable $x_i.j$, where $1 \leq i \leq n$ and $1 \leq j \leq |\sigma(s_i)|$ is a position in $\sigma(s_i)$. $x_i.j$ is called a *defining position*, if

$\delta_r(i) \neq \delta_\sigma(s_i, j)$.[13] $x_i.j$ is called an *applied position*, otherwise. Semantic rules will be allowed to refer to *auxiliary* attributes, too. These will be denoted as $x_i.j$, with $|\sigma(s_i)| < j$. For auxiliary attributes no distinction is made with respect to *in* and *out*. Thus, positions of auxiliary attributes are both defining and applied.[14]

- An *attribute rule* is a formula of one of the following two kinds:
  - $x_{i_0}.j_0 = f(x_{i_1}.j_1, \ldots, x_{i_k}.j_k)$,[15]

    where $f \in \Omega'$, $x_{i_0}.j_0$ is a defining and the $x_{i_l}.j_l$, $l \geq 1$, are applied occurrences of attributes,
  - $r(x_{i_1}.j_1, \ldots, x_{i_n}.j_n)$,[16]

    where $r \in R'$, and where any of the $x_{i_l}.j_l$ is a defining position of an attribute, if and only if $\delta_r(l) = in$.

The rule is said to define $x_0.j_0$ and the $x_{i_l}.j_l$ for which $\delta_r(l) = in$, respectively. A rule of the first kind is called a *functional rule* as opposed to the *relational rules* of the second kind.

Definition 5.1. σ is called a *language morphism*, if
- $s \in SM$ implies $\sigma(s) \in SM'$ and $\delta_\sigma(s, 1) = out$,

and if for any syntactic rule p, $\sigma(p)$ is a conjunction of attribute rules for p such that
- for any defining position of an attribute there exists exactly one attribute rule of p that defines this position;
- any applied position of an attribute of a syntactic sort $s' \in S'$ occurs in exactly one *relational* rule of p that applies (i.e. does not define) the position.

The first condition says that semantic sorts are translated into semantic sorts. The second condition guarantees that the system of attribute rules for a syntactic relation symbol is consistent and complete. The last condition assures that the dependencies between attributes of syntactic sorts are complete graphs in the above sense.

This restricted class of signature morphisms is still somewhat wider than the class of attribute grammars.

Observation 5.1. A language morphism is an attribute grammar in the classical sense, iff
- each attribute rule is a functional rule,
- each attribute is of a semantic sort,
- each relation symbol does have only syntactic parameters.

[13] Defining occurrences are the inherited attributes of right-side positions and the synthesized attributes of (the) left-side position(s).

[14] In the examples we have written simply x_i for $x_i.1$, if $|\sigma(s_i)| = 1$. Also, auxiliary variables have just been denoted by new identifiers.

[15] This is supposed to also include equations of form $x_i.j = x_k.l$ for attributes of a *semantic* sort.

[16] This is supposed to also include equations of form $x_i.j = x_k.l$ for attributes of a *syntactic* sort. Here, the left side is assumed to be an *in*-position and the right side an *out*-position of the 'identity relation' $r \equiv \_ = \_$.

In the next section we will indicate that the concept of attribute evaluation can be generalized to language morphisms:
– attribute evaluation according to relational rules will amount to output a graph that represents the relational dependencies between the involved syntactic attributes;
– semantic parameters of relation symbols are considered as to represent lexical information about terminals.

For now we give some illustration using our examples of Sections 3 and 4. All signature morphisms defined there are language morphisms in the above sense. The following example is taken from the definition of ρ_{L1}.

Example 5.1

Block, Stat $\mapsto$ (*env*: *Bindings, memory*: *MemUnit, code*: *Sstat*)
Decl $\mapsto$ (*env*: *Bindings, memory*: *MemUnit*)
...
BLOCK → **begin** DECL ; STAT **end** $\mapsto$
BLOCK.*env* → scope with binding constructs C
C → DECL.*env* ; STAT.*env*
BLOCK.*memory* → segment consisting of U
U → DECL.*memory* concatenated STAT.*memory*
BLOCK.*code* = STAT.*code*
...

In the examples, the argument positions left of "→" are the *in*-positions of a relation symbol. Conversely, the *out*-positions are the argument positions right of "→". In the above example, only synthesized attributes occur, i.e. any of the positions *env, memory,* and *code* is an *out*-position of the representation of *Block, Stat,* and *Decl.* Also, each sort is a syntactic sort. So, the language morphism properties require the existence of exactly one attribute rule for each defining occurrence (i.e. for BLOCK.*env* C, BLOCK.*memory*, U, and BLOCK.*code*) and the occurrence of any (syntactic) applied attribute position (i.e. C, DECL.*env*, STAT.*env*, U, DECL.*memory*, STAT.*memory*, and STAT.*code*) in exactly one relational rule. Now, all attribute rules associated with the block rule are relational rules. E.g. C → DECL.*env*;STAT.*env* defines C and applies DECL.*env* and STAT.*env*. Here, C is an auxiliary attribute of a syntactic sort that, as required, is applied exactly once namely in the first attribute rule.

Let us consider a second example taken from the definition of ρ_C.

Example 5.2

Sstat $\mapsto$ (*iLabCtr, sLabCtr*: *Integer, c*: *Tstat*)
...
s0 → s1;s2 $\mapsto$
s1.*iLabCtr* = s0.*iLabCtr*
s2.*sLabCtr* = s1.*sLabCtr*
s0.*sLabCtr* = s2.*sLabCtr*
s0.*c* → s1.*c*;s2.*c*

Here, *Tstat* and *Sstat* are the syntactic sorts, *Integer* is a semantic sort. *iLabCtr* is an inherited attribute, i.e. an *in*-position in $\rho_C(Sstat)$, whereas *sLabCtr* and *c* are synthesized, i.e. *out*-positions. Then the attribute rules for s0 → s1;s2 have to contain exactly one rule to define s0.*c*, s0.*sLabCtr*, s1.*iLabCtr*, and s2.*iLabCtr*. Obviously, this is the case. Moreover, s1.*c* and s2.*c* have to occur applied in exactly one relational rule. The rule s0.*c* → s1.*c*;s2.*c* achieves this.

Theorem 5.1. (1) *The identity is a language morphism.*
(2) *$\sigma'\sigma$ is a language morphism, provided σ and σ' are.*

Proof. (1). Set $\delta_\sigma(s, 1) = out$, for any $s \in S$. Then, the assertion follows immediately.
(2) Let $\sigma'' = \sigma'\sigma$. Set, for $s \in S$,

$$\delta_{\sigma''}(s, i.j) = \textbf{if } \delta_\sigma(s, i) = \delta_{\sigma'}(s_i, j) \textbf{ then } out \textbf{ else } in.$$

(Here and in the following, $i.j$ denotes the index $\sum_{l=1}^{i-1} |\sigma'(s_l)| + j$ in $\sigma''(s)$, if $\sigma(s) = s_1 \cdots s_n$, $1 \leq i \leq n$, $1 \leq j \leq |\sigma'(s_i)|$.) If $s \in SM$, then $\sigma''(s) = \sigma'(\sigma(s)) \in SM''$ and $\delta_{\sigma''}(s, 1) = \delta_{\sigma''}(s, 1.1) = out$.

We show now that any definining attribute position has exactly one attribute rule. Let $q = r(x_1{:}s_1, \ldots, x_n{:}s_n)$ be a syntactic rule, let $\sigma(s_i) = s_{i1} \ldots s_{ik_i}$, $\sigma'(s_{ij}) = s_{ij1} \ldots s_{ijl_{ij}}$. Then, $x_i.j.p$ is a defining position in q according to σ'', if $\delta_r(i) \neq \delta_{\sigma''}(s_i, j.p)$.

Case 1: $\delta_r(i) = out$. Then, $\delta_{\sigma''}(s_i, j.p) = in$, thus $\delta_\sigma(s_i, j) \neq \delta_{\sigma'}(s_{ij}, p)$.

Case 1.1: $\delta_\sigma(s_i, j) = out$ and $\delta_{\sigma'}(s_{ij}, p) = in$. Then, $x_i.j$ is an applied position in q and is of a syntactic sort $s_{ij} \in SY'$. Thus, there exists exactly one relational attribute rule $q' = r'(y_1, \ldots, y_m)$ in $\sigma(q)$ in which $x_i.j$ occurs exactly once, say at i_0. Therefore, $\delta_{r'}(i_0) = out$. As $\delta_{\sigma'}(s_{ij}, p) = in$, $\sigma'(q')$ defines exactly one attribute rule q'' for y_{i_0} $(= x_i.j.p)$. Thus, $\sigma''(q)$ defines the rule q'' for $x_i.j.p$. The uniqueness of this rule follows from the uniqueness of q' and the uniqueness of q'' in $\sigma'(q')$.

Case 1.2: $\delta_\sigma(s_i, j) = in$ and $\delta_{\sigma'}(s_{ij}, p) = out$. Then, $x_i.j$ is a defining position in q. There exists exactly one corresponding rule q' in $\sigma(q)$. If $s_{ij} \in SM'$, then $\sigma'(q')$ is the rule for $x_i.j.p$. Otherwise, q' must be a relational rule (the result sorts of operators are semantic). If $q' = r'(y_1, \ldots, y_k)$, then $x_i.j = y_{i_0}$, for some i_0. It holds $\delta_r(i_0) = in \neq \delta_{\sigma'}(s_{ij}, p)$, hence $x_i.j.p$ is a defining position in q' according to σ'. Then, $\sigma'(q')$ (which is contained in $\sigma''(q)$) contains a rule for $x_i.j.p$.

Case 2: $\delta_r(i) = in$. By symmetry.

It remains to be shown that for any applied position $x_i.j.p$ in q (q as above) of a syntactic sort occurs in exactly one relational rule of σ''.

First we note that $\delta_r(i) = \delta_{\sigma''}(s_i, j.p)$ in this case.

Case 1: $\delta_r(i) = out$. Then, $\delta_{\sigma''}(s_i, j.p) = out$, thus $\delta_\sigma(s_i, j) = \delta_{\sigma'}(s_{ij}, p)$.

Case 1.1: $\delta_\sigma(s_i, j) = out$. Then, $x_i.j$ is an applied position in q of a syntactic sort. There exists, therefore, a unique relational rule q' in $\sigma(q)$ in which $x_i.j$ occurs. This occurrence is at an *out*-position implying that $x_i.j.p$ is an applied position in q'. Again, there exists, then, a unique relational rule q'' for q' that applies $x_i.j.p$. Together, q'' is the unique relational rule for q according to σ'' that applies $x_i.j.p$.

Case 1.2: $\delta_\sigma(s_i, j) = in$. Here, $x_i.j$ is a defining position in q. Thus, there exists exactly one rule q' for q defining $x_i.j$ according to σ. As $x_i.j$ is of a syntactic sort, q' has to be a relational rule. Moreover, as q' is defining for $x_i.j$, i.e. occurs at an *in*-position in q', $\delta_{\sigma'}(s_{ij}, p) = in$, too. Hence, there exists exactly one occurrence of $x_i.j.p$ as an applied position in some relational σ'-rule q'' in $\sigma(q')$. The latter is, then, the unique applied occurrence in a σ''-rule of q.

Case 2: By symmetry. □

The next theorem asserts that language morphisms are closed under parameter passing, i.e. the existence of certain pushout diagrams.

Theorem 5.2. *Given σi: $\Sigma \to \Sigma 1$, $i = 1,2$, two signature morphisms such that $\Sigma \subseteq \Sigma 1$ and $\sigma 1$ is the inclusion morphism. Consider the pushout diagram*

$$\begin{tikzcd} \Sigma \arrow[r, "\sigma 1"] \arrow[d, "\sigma 2"'] & \Sigma 1 \arrow[d, "\sigma' 2"] \\ \Sigma 2 \arrow[r, "\sigma' 1"] & \Sigma_{po} \end{tikzcd}$$

If $\sigma 2$ is a language morphism, then $\sigma' 1$ and $\sigma' 2$ are language morphisms, too.

Proof. We repeat the definition of Σ_{po} and of the $\sigma' i$ from the proof of Theorem 4.3 about pushouts;

$S_{po} = (S1 - S) + S2$
$\sigma'2(s) = \textbf{if } s \in S \textbf{ then } \sigma 2(s) \textbf{ else } s$
$\Omega_{po} = \Omega 2 + \Omega 3,$
  where $\Omega 3 = \{f.i{:}\sigma'2(u)s_i \mid f{:}us \in \Omega 1 - \Omega,\ \sigma'2(s) = s_1 \cdots s_n,\ 1 \leq i \leq n\}$
$\sigma'2(f) = \textbf{if } f{:}us \in \Omega 1 - \Omega, \sigma'2(s) = s_1 \cdots s_n \textbf{ then } (f.1, \ldots, f.n) \textbf{ else } \sigma 2(f)$
$R_{po} = R2 + R3$ where $R3 = \{r{:}\sigma'2(u) \mid r{:}u \in R1 - R\}$
$\sigma'2(r) = \textbf{if } r \in R \textbf{ then } \sigma 2(r) \textbf{ else } r$
$\sigma'1$ the inclusion $\Sigma 2 \subseteq \Sigma_{po}$.

It remains to specify the sort classification of the $\sigma' i$ and the classification of the parameter positions of the relation symbols in R_{po}. We define, for $\sigma'1$,

$\delta_{\sigma'1}(s, 1) = out$ for $s \in S2$.

(Note that $\sigma'2$ is an inclusion, i.e. $\sigma'2(s) \in S_{po}$.) For $\sigma'2$, we define

$\delta_{\sigma'2}(s, 1) = out$ for $s \in S1 - S$,
$\delta_{\sigma'2}(s, i) = \delta_{\sigma 2}(s, i)$ for $s \in S$ and all i.

Finally, for S_{po} we set

$$SY_{po} = SY2 + (SY1 - SY), \qquad SM_{po} = SM2 + (SM1 - SM).$$

The directions of the arguments of R_{po}-relation are as follows:

$$\delta_{\sigma'1(r)}(i) = \delta_r(i) \quad \text{for } r \in R2,$$
$$\delta_{\sigma'2(r)}(i.j) = \textbf{if } \delta_r(i) = in \textbf{ then } \delta_{\sigma 2}(s_i, j) \textbf{ else } \neg\delta_{\sigma 2}(s_i, j),$$
$$\text{if } r: s_1 \cdots s_n \in (R1 - R),\ 1 \leq i \leq n,\ 1 \leq j \leq |\sigma'2(s_i)|.$$

Here, the index $i.j$ is given as in the proof of the last theorem. The assertion of the theorem is an immediate consequence of these definitions and the definition of a language morphism. □

5.3. Generation of attribute evaluators for language morphisms

The theorems of the last section have stated that language morphisms are closed under applying the combinators **apply** and **abstract** in the following sense:

Corollary 5.1. *Let*

$$T(P_1, \ldots, P_n, \sigma_1, \ldots, \sigma_m)$$

be a term in parameterized data types P_i, in language morphisms σ_j, and in the combinators **apply** *and* **abstract.** *If, after applying Theorem* 4.5, *this term can be transformed into an equivalent term*

$$\textbf{abstract}(T'(P_1, \ldots, P_n, \sigma'_1, \ldots, \sigma'_k), \sigma),$$

then any of the σ'_j as well as σ are again language morphisms.

This result suggests the following treatment of modular compiler descriptions in a compiler generating system:

The compiler generating system accepts as parameters in **apply** and **abstract** terms language morphisms only. The language morphism property can be checked automatically.

The corollary guarantees that any combination of the so described compiler modules is of form

$$\Sigma \to \Sigma 1 \xrightarrow{\sigma} \Sigma 2$$

where $\Sigma \subseteq \Sigma 1$ is the parameterized data type that results from combining all predefined data types by an **apply**-term, and where σ is a language morphism.

For language morphisms, attribute evaluators can be generated under the following additional assumptions:

– Any operator of the predefined data types has only parameters of semantic sorts and can, thus, be called like a function of a, say, PASCAL-package of procedure and type definitions.

– Relational attribute rules are handled as if the relation symbol was a function constructing the graph of relational dependencies between attributes of a syntactic type. This graph can be viewed as the target program that is the final output of the

attribute evaluation process. This target program will usually represent an intermediate code, subject to later machine code generation, cf. Section 6.
– Any semantic parameter of a relation symbol in $\Sigma 1$ is considered as lexical information about a terminal symbol. The latter is assumed to be provided by the lexical analyser.

6. Other applications

The principal idea behind the modularization concept was to base the description of compiler modules on specifically tailored abstractions of the concrete syntax of the language. As a consequence, there exist different syntactic representations of the source programs on the level of compiler descriptions. The combinators **apply** and **abstract** allow to adapt modules to new syntactic environments. In this section we demonstrate how this can be utilized to describe transformations of source program representations at compile time.

6.1. Relating concrete and abstract syntax

The concrete parse tree of a source program is its most detailed syntactic representation. It is usually too big and contains too much irrelevant structure to be actually built up during compilation. Most systems that can generate multi-pass compilers, therefore, provide for constructing an abstract tree instead. Some compiler generating systems automatically cut the tree where only trivial semantic rules are associated [27]. The compiler describer has to associate the semantic rules with the concrete syntax. Others [15] allow the user to explicitly specify the abstract syntax using special description tools such as string-to-tree grammars. This has the advantage that semantic parts of the compiler description can be based on the abstract syntax, achieving some independence from the parsing technique and reducing the amount of trivial semantic rules.

In our approach, the transformation of the concrete into the abstract parse tree can easily be described as a signature morphism. Figure 12 shows part of a signature morphism ρ_{CA} that sends the concrete syntax $L1_C$ of L1 into the abstract syntax that is the basis for *L1Comp*. The abstract syntax is ambiguous, e.g. EXP→ EXP *op* EXP, the concrete is not. It contains less (chain) productions, nonterminals, delimiters, and keywords. This results in

$$\begin{aligned} L1_C\text{-}Comp &= \mathbf{abstract}(Predefined,\ \rho_{CA}\rho'_{L1}) \\ &= \mathbf{abstract}(\mathbf{abstract}(Predefined,\ \rho'_{L1}),\ \rho_{CA}). \end{aligned}$$

The first variant would correspond to an adaptation of the compiler description to the concrete syntax. This process takes place at description time. The second variant views the compiler as a sequence of two translation steps. The first is syntactic analysis and the construction of the abstract parse tree. The second step is the rest of the compilation process. The first variant would be chosen for a one-pass compiler.

...

StatList	$\mapsto$ *Statement*	λ STATS, STATS0, STATS1
WhileStat	$\mapsto$ *Statement*	λ WHILE
Stat	$\mapsto$ *Statement*	λ STAT
Exp	$\mapsto$ *Exp*	λ E, E0, E1
Term	$\mapsto$ *Exp*	λ T, T0, T1
Factor	$\mapsto$ *Exp*	λ F
$Id	$\mapsto$ *Id*	λ ID

...

...

...

STATS0 $\to$ STATS1;STAT	$\mapsto$ STATS0 $\to$ STATS1;STAT
STATS $\to$ STAT	$\mapsto$ STATS = STAT
STAT $\to$ WHILE	$\mapsto$ STAT = WHILE

...

E0 $\to$ E1 + T	$\mapsto$ E0 $\to$ E1 + T
E $\to$ T	$\mapsto$ E = T
T0 $\to$ T1 * F	$\mapsto$ T0 $\to$ T1 * F
T $\to$ F	$\mapsto$ T = F
F $\to$ (E)	$\mapsto$ F = E
F $\to$ ID	$\mapsto$ F $\to$ ID

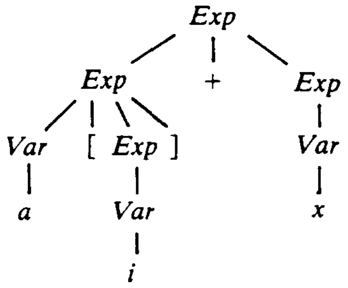
Fig. 12. Compiler for concrete L1.

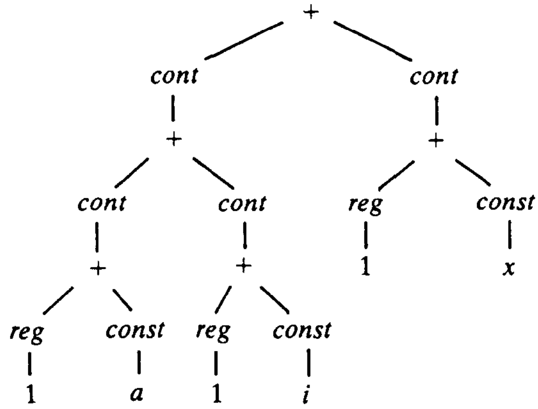
Here the distinction between different syntactic environments is merely conceptual. The second variant would increase efficiency in a multi-pass compiler as the abstract parse tree would be the intermediate form that is to be explored.

6.2. Interface to machine code generation

Following [28], code selection is viewed as the searching of the intermediate program representation for patterns that can be coded into corresponding target machine instructions. Subsequent approaches [13, 18], have utilized LR-parser generators to generate code generators from machine descriptions that relate pieces of machine code to patterns of intermediate language constructs. It is obvious that this purely syntactic treatment requires as much semantic information to be encoded syntactically as possible.

Example 6.1. Consider the L1-expression $a[i]+x$. The left tree below is its abstract parse tree. However, a tree that is suitable for code generation would have to have a shape like the right tree. (It is assumed that for the array variable a a dope vector allocation scheme is to be used and that variables are addressed via offsets to an address kept in register R1.)





The example demonstrates that the trees from which code is to be generated are different from abstract parse trees.

[14] and [18] apply this code generation concept to a fixed source language such that the generation of the intermediate program is performed by a hand-written compiler frontend. MUG2 [15] is the only compiler generating system that provides a tool for describing syntactic transformations of the intermediate program representation. This allows, among others, for specifying the transformation of the abstract parse tree into an intermediate form suitable for machine code generation. However, this part of MUG2 has not been implemented yet so that no practical results have been obtained so far.

In the context of viewing a compiler as a language morphism, a different idea is quite obvious: The constructs of the target language can be represented as relation symbols. This corresponds to our definition of *TargetLang*. Then, the handling of syntactic attributes during attribute evaluation, cf. Section 5.3 amounts to building up the graphs which the values of the syntactic attributes denote.

Example 6.2. For the above expression e, attribute evaluation in the L1-compiler yields a relation expression of form

$$\begin{array}{l}
E0 \rightarrow E1 + E2 \wedge \\
\quad E1 \rightarrow \mathit{cont}(E3) \wedge \\
\quad\quad E3 \rightarrow E4 + E5 \wedge \\
\quad\quad\quad E4 \rightarrow E6 + E7 \wedge \\
\quad\quad\quad\quad E6 \rightarrow \mathit{register}(1) \wedge \\
\quad\quad\quad\quad E7 \rightarrow \mathit{const}(a) \wedge \\
\quad\quad\quad E5 \rightarrow E8 + E9 \wedge \\
\quad\quad\quad\quad E8 \rightarrow \mathit{register}(1) \wedge \\
\quad\quad\quad\quad E9 \rightarrow \mathit{const}(i) \wedge \\
\quad E2 \rightarrow \mathit{cont}(E10) \wedge \\
\quad\quad E10 \rightarrow E11 + E12 \wedge \\
\quad\quad\quad E11 \rightarrow \mathit{register}(1) \wedge \\
\quad\quad\quad E12 \rightarrow \mathit{const}(x)
\end{array}$$

which, if written as a tree (the logical variables Ei representing its nodes), has exactly the form as required in the last example.

7. Conclusions

The main contribution of this paper is the introduction of a concept for modular definition of software for language implementation. The basic idea was to employ relations to characterize syntactic contexts of constructs in a program. The relation symbols can be viewed as defining an abstract syntax that is specifically tailored to the module being defined. Thereby, modules become independent of the concrete (syntax of the) language. Modules in our sense encapsulate implementation decisions that correspond to fundamental semantic concepts and compiling techniques, such as binding concept, control constructs, type concept. This increases flexibility of language implementation considerably.

The main technical achievements are due to the formal system which we employ for specifying the implementation of relations. Rather than adopting a general logical framework such as the one provided by PROLOG, we introduced an extended version of the concept of a signature morphism which is the basic formal tool of known approaches to structuring specifications of abstract data types. As we have demonstrated, this allows to apply basic results of abstract data type theory about the structuring and parameterizing data types: Basic compiler modules can be defined according to the specific language concepts a compiler has to deal with. Then, the modules can be combined (using **apply** and **abstract**) such that they, finally, make the complete compiler for the concrete language. Secondly, and this is important from a practical point of view, we have indicated that attribute grammars are a particular subclass of such signature morphisms. This way we are able to apply the structuring techniques to compiler descriptions as they are input to today's compiler generating systems. In such applications one has to find ways to guarantee that a compiler description which has been combined out of library modules can in fact be viewed as an attribute grammar, as it has been the case for our sample compilers. A solution to this problem has been provided by the notion of a language morphism. Language morphisms form a subclass of signature morphisms that is closed under the combinators we use and is at the same time an 'algebraic variant' of attribute grammars.

We have, thus, also given a new algebraic view of attribute grammars, different from that of [9]. Whereas in the latter paper attributes are functionalized into attribute dependencies to obtain denotational semantics definitions in the sense of [4], we have modelled the process of *attribute evaluation* algebraically. This corresponds to what a compiler module does, namely decorate intermediate forms with semantic information.

This paper has only dealt with aspects of modular *implementation* of compiler modules. An open problem is how to *specify* the abstract properties that a module is required to have. 'How' means that the specification language should have 'nice' properties. (At the end of Section 2 we have specified a property of *identification* as a first order formula.) The properties should allow to obtain results about the existence of (persistent) type constructors for parameterized specifications of relations and to investigate their correspondence to formal implementations of the kind introduced above.

Another aspect that has not investigated in full detail yet concerns the problem of compiler correctness. One of the important goals that modularization is supposed to achieve is that the structure of program correctness proofs can be chosen according to the module structure of the program: first the modules are proven correct separately and, then, the correctness proof of the program is obtained by combining the proofs. In fact, formal investigations of this problem have substantiated this claim, cf. [11, 12, 14, 23], among others. We believe that the specification of the semantics of programming languages can be derived in exactly the same way as shown in this paper for compilers. Data types that specify the *semantics* of the language facets involved are combined into the definition of the semantics for the particular language. Then, the correctness of its compiler follows from the correctness of its modules.

Appendix A. Expanded version of the L1-compiler

Block, Statement $\mapsto$
    (*env.iSt* : *StStates*, *env.sSt* : *StStates*),
    (*memory.iTop*, *memory.sTop*, *memory.offset*, *memory.size* : *Integer*),
    (*code.iLabCtr*, *code.sLabCtr* : *Integer*, *code.c* : *Tstat*))
Decl $\mapsto$
    ((*env.iSt* : *StStates*, *env.sSt* : *StStates*),
    (*memory.iTop*, *memory.sTop*, *memory.offset*, *memory.size* : *Integer*))
\$*Id* $\mapsto$ *Id*
Var $\mapsto$
    ((*env.iSt* : *StStates*, *env.sSt* : *StStates*),
    code : *Texp*)
Exp, Bexp $\mapsto$
    ((*env.iSt* : *StStates*, *env.sSt* : *StStates*),
    code : *Texp*)

\$*eq*(*Id*, *Id*) : *Bool* $\mapsto$ *eq*(*id*, *id*) : *Bool*
---
prog $\to$ BLOCK $\mapsto$
    BLOCK.*env.iSt* = *init*
    BLOCK.*memory.iTop* = 0
    BLOCK.*memory.offset* = 0
    BLOCK.*code.iLabCtr* = 0

BLOCK → **begin** DECL ; STAT **end** ↦
    C.iSt = enterScope (BLOCK.env.iSt)
    BLOCK.env.sSt = leaveScope(C.sSt)
    DECL.env.iSt = C.iSt
    STAT.env.iSt = DECL.env.sSt
    C.sSt = STAT.env.sSt
    U.iTop = BLOCK.memory.iTop
    BLOCK.memory.size = U.size
    U.offset = 0
    BLOCK.memory.sTop = U.sTop
    DECL.memory.offset = U.offset
    STAT.memory.offset = U.offset + DECL.memory.size
    DECL.memory.iTop = U.iTop
    STAT.memory.iTop = DECL.memory.sTop
    U.sTop = STAT.memory.sTop
    U.size = DECL.memory.size + STAT.memory.size
    STAT.code.iLabCtr = BLOCK.code.iLabCtr
    BLOCK.code.sLabCtr = STAT.code.sLabCtr
    BLOCK.code.c = STAT.code.c
DECL0 → DECL1 ; DECL2 ↦
    DECL1.env.iSt = DECL0.env.iSt
    DECL2.env.iSt = DECL1.env.sSt
    DECL0.env.sSt = DECL2.env.sSt
    DECL1.memory.offset = DECL0.memory.offset
    DECL2.memory.offset = DECL0.memory.offset + DECL1.memory.size
    DECL1.memory.iTop = DECL0.memory.iTop
    DECL2.memory.iTop = DECL1.memory.sTop
    DECL0.memory.sTop = DECL2.memory.sTop
    DECL0.memory.size = DECL1.memory.size + DECL2.memory.size
DECL → ID:**integer** ↦
    DECL.env.sSt = enter(DECL.env.iSt,ID,DECL.memory.iTop)
    DECL.memory.sTop = DECL.memory.iTop + 1
    DECL.memory.size = 1
STAT → BLOCK ↦
    BLOCK.env.iSt = STAT.env.iSt
    STAT.env.sSt = BLOCK.env.sSt
    BLOCK.memory.offset = STAT.memory.offset
    BLOCK.memory.iTop = STAT.memory.iTop
    STAT.memory.sTop = BLOCK.memory.sTop
    STAT.memory.size = BLOCK.memory.size
    BLOCK.code.iLabCtr = STAT.code.iLabCtr
    STAT.code.sLabCtr = BLOCK.code.sLabCtr
    STAT.code.c = BLOCK.code.c

STAT0 → STAT1 ; STAT2 ↦
  STAT1.*env.iSt* = STAT0.*env.iSt*
  STAT2.*env.iSt* = STAT1.*env.sSt*
  STAT0.*env.sSt* = STAT2.*env.sSt*
  STAT1.*memory.offset* = STAT0.*memory.offset*
  STAT2.*memory.offset* = STAT0.*memory.offset*
  STAT1.*memory.iTop* = STAT0.*memory.iTop*
  STAT2.*memory.iTop* = STAT0.*memory.iTop*
  STAT0.*memory.sTop* = *max*(STAT1.*memory.sTop*, STAT2.*memory.sTop*)
  STAT0.*memory.size* = *max*(STAT1.*memory.size*,STAT2.*memory.size*)
  STAT1.*code.iLabCtr* = STAT0.*code.iLabCtr*
  STAT2.*code.iLabCtr* = STAT1.*code.sLabCtr*
  STAT0.*code.sLabCtr* = STAT2.*code.sLabCtr*
  STAT0.*code.c* → STAT1.*code.c* ; STAT2.*code.c*
STAT → V := EXP ↦
  STAT.*memory.sTop* = STAT.*memory.iTop* + 0
  STAT.*memory.size* = 0
  V.*env.iSt* = STAT.*env.iSt*
  EXP.*env.iSt* = V.*env.sSt*
  STAT.*env.sSt* = EXP.*env.sSt*
  STAT.*code.sLabCtr* = STAT.*code.iLabCtr*
  STAT.*code.c* → assign(V.*code*, EXP.*code*)
V → ID ↦
  V.*env.sSt* = V.*env.iSt*
  V.*code* → register(1) + const(*lookup*(V.*env.iSt*, ID))
EXP → I ↦
  EXP.*env.sSt* = EXP.*env.iSt*
  EXP.*code* → const(I)
EXP → V ↦
  V.*env.iSt* = EXP.*env.iSt*
  EXP.*env.sSt* = V.*env.sSt*
  EXP.*code* → cont(V.*code*)
EXP0 → EXP1 *op* EXP2 (*op* = +, −, *, /) ↦
  EXP1.*env.iSt* = EXP0.*env.iSt*
  EXP2.*env.iSt* = EXP1.*env.sSt*
  EXP0.*env.sSt* = EXP2.*env.sSt*
  EXP0.*code* → EXP1.*code op* EXP2.*code*
STAT0 → **while** BEXP **do** STAT1 ↦
  BEXP.*env.iSt* = STAT0.*env.iSt*
  STAT1.*env.iSt* = BEXP.*env.sSt*
  STAT0.*env.sSt* = STAT1.*env.sSt*
  STAT1.*memory.offset* = STAT0.*memory.offset*
  STAT1.*memory.iTop* = STAT0.*memory.iTop*

STAT0.*memory.sTop* = STAT1.*memory.sTop*
STAT0.*memory.size* = STAT1.*memory.size*
STAT0.*code.c* → label(STAT0.*code.iLabCtr*);
    JifF(BEXP.*code.* const(STAT0.*code.iLabCtr* + 1));
    STAT1.*code.c*;
    Jmp(const(STAT0.*code.iLabCtr*));
    label(STAT0.*code.iLabCtr* + 1)
STAT1.*code.iLabCtr* = STAT0.*code.iLabCtr* + 2
STAT0.*code.sLabCtr* = STAT1.*code.sLabCtr*

BEXP → EXP1 *relop* EXP2 (*relop* = ⟨,⟩, =, . . .) ↦
EXP1.*env.iSt* = BEXP.*env.iSt*
EXP2.*env.iSt* = EXP1.*env.sSt*
BEXP.*env.sSt* = EXP2.*env.sSt*
BEXP.*code* → EXP1.*code relop* EXP2.*code*

STAT0 → **if** BEXP **then** STAT1 ↦
BEXP.*env.iSt* = STAT0.*env.iSt*
STAT1.*env.iSt* = BEXP.*env.sSt*
STAT0.*env.sSt* = STAT1.*env.sSt*
STAT1.*memory.offset* = STAT0.*memory.offset*
STAT1.*memory.iTop* = STAT0.*memory.iTop*
STAT0.*memory.sTop* = STAT1.*memory.sTop*
STAT0.*memory.size* = STAT1.*memory.size*
STAT0.*code.c* → JifF(BEXP.*code*,const(STAT0.*code.iLabCtr*));
    STAT1.*code.c*;
    Jmp(const(STAT0.*code.iLabCtr* + 1));
    label(STAT0.*code.iLabCtr*);
    S.c;
    label(STAT0.*code.iLabCtr* + 1)
STAT1.*code.iLabCtr* = STAT0.*code.iLabCtr* + 2
S.iLabCtr = STAT1.*code.sLabCtr*
STAT0.*code.sLabCtr* = *S.sLabCtr*
S.sLabCtr = *S.iLabCtr*
S.c → skip

Appendix B. Expanded version of the L2-compiler

Stat ↦
(*env.iSt* : *StStates*, *env.sSt* : *StStates*),
(*code.iLabCtr*, *code.sLabCtr* : *Integer*, *code.c* : *Tstat*))

Decl, *Field* ↦
((*env.iSt* : *StStates*, *env.sSt* : *StStates*),
(*memory.iTop*, *memory.sTop*, *memory.offset*, *memory.size* : *Integer*))

TypeDen ↦
  ((*env.iSt* : *StStates*, *env.sSt* : *StStates*),
  (*memory.iTop*, *memory.sTop*, *memory.offset*, *memory.size* : *Integer*),
  type : *TypeDenotation*)
$Id ↦ *Id*
Var ↦
  ((*env.iSt* : *StStates*, *env.sSt* : *StStates*),
  code : *Texp*,
  recordNmb : *Integer*)
Exp ↦
  ((*env.iSt* : *StStates*, *env.sSt* : *StStates*),
  code : *Texp*)

$eq(Id, Id) : Bool ↦ *eq*(*Id*, *Id*) : *Bool*
---
program → **begin** DECL;STAT **end** ↦
  C.iSt = *init*
  C'.iSt = *enterScope*(*C.iSt*)
  C.sSt = *leaveScope*(*C'.sSt*)
  DECL.*env.iSt* = *C'.iSt*
  STAT.*env.iSt* = DECL.*env.sSt*
  C'.sSt = STAT.*env.sSt*
  DECL.*memory.iTop* = 0
  DECL.*memory.offset* = 0
  STAT.*code.iLabCtr* = 0
DECL0 → DECL1 ; DECL2 ↦
  DECL1.*env.iSt* = DECL0.*env.iSt*
  DECL2.*env.iSt* = DECL1.*env.sSt*
  DECL0.*env.sSt* = DECL2.*env.sSt*
  DECL1.*memory.offset* = DECL0.*memory.offset*
  DECL2.*memory.offset* = DECL0.*memory.offset* + DECL1.*memory.size*
  DECL1.*memory.iTop* = DECL0.*memory.iTop*
  DECL2.*memory.iTop* = DECL1.*memory.sTop*
  DECL0.*memory.sTop* = DECL2.*memory.sTop*
  DECL0.*memory.size* = DECL1.*memory.size* + DECL2.*memory.size*
DECL → **type** ID = TYPEDEN ↦
  TYPEDEN.*env.iSt* = DECL.*env.iSt*
  D.iSt = TYPEDEN.*env.sSt*
  DECL.*env.sSt* = *D.sSt*
  D.sSt = *enter*(*D.iSt*, ID, *mkTypeIdInfo*(TYPEDEN.*memory.size*,
    TYPEDEN.*type*))
  DECL.*memory.sTop* = DECL.*memory.iTop* + 0
  DECL.*memory.size* = 0

TYPEDEN.*memory.iTop* = 0
TYPEDEN.*memory.offset* = 0

DECL → **var** ID:TYPEDEN ↦
TYPEDEN.*env.iSt* = DECL.*env.iSt*
D.iSt = TYPEDEN.*env.sSt*
DECL.*env.sSt* = *D.sSt*
D.sSt = *enter*(*D.iSt*, ID, *mkVarInfo*(*decl.memory.iTop*, TYPEDEN.*type*))
TYPEDEN.*memory.offset* = DECL.*memory.offset*
TYPEDEN.*memory.iTop* = DECL.*memory.iTop*
DECL.*memory.sTop* = TYPEDEN.*memory.sTop*
DECL.*memory.size* = TYPEDEN.*memory.size*

TYPEDEN → **integer** ↦
TYPEDEN.*type* = *intType*
TYPEDEN.*memory.sTop* = TYPEDEN.*memory.iTop* + 1
TYPEDEN.*memory.size* = 1

TYPEDEN → **record** FIELD **end** ↦
FIELD.*env.iSt* = *enterScope*(TYPEDEN.*env.iSt*)
TYPEDEN.*env.sSt* = *leaveScope*(FIELD.*env.sSt*)
FIELD.*memory.iTop* = TYPEDEN.*memory.iTop*
TYPEDEN.*memory.size* = FIELD.*memory.size*
FIELD.*memory.offset* = 0
TYPEDEN.*memory.sTop* = FIELD.*memory.sTop*
TYPEDEN.*type* = *recordType*(*currentScope*(TYPEDEN.*env.iSt*))

TYPEDEN → ID ↦
TYPEDEN.*type* = *getType*(*lookup*(TYPEDEN.*env.iSt*,ID)
TYPEDEN.*memory.sTop* = TYPEDEN.*memory.iTop* + SIZE
TYPEDEN.*memory.size* = SIZE
SIZE = *getSize*(*lookup*(TYPEDEN.*env.iSt*, ID))

FIELD0 → FIELD1 ; FIELD2 ↦
FIELD1.*env.iSt* = FIELD0.*env.iSt*
FIELD2.*env.iSt* = FIELD1.*env.sSt*
FIELD0.*env.sSt* = FIELD2.*env.sSt*
FIELD1.*memory.offset* = FIELD0.*memory.offset*
FIELD2.*memory.offset* = FIELD0.*memory.offset* + FIELD1.*memory.size*
FIELD1.*memory.iTop* = FIELD0.*memory.iTop*
FIELD2.*memory.iTop* = FIELD1.*memory.sTop*
FIELD0.*memory.sTop* = FIELD2.*memory.sTop*
FIELD0.*memory.size* = FIELD1.*memory.size* + FIELD2.*memory.size*

FIELD → ID : TYPEDEN ↦
TYPEDEN.*env.iSt* = FIELD.*env.iSt*
D.iSt = TYPEDEN.*env.sSt*
FIELD.*env.sSt* = *D.sSt*
D.sSt = *enter*(*D.iSt*, ID, *mkVarInfo*(FIELD.*memory.offset*, TYPEDEN.*type*))

TYPEDEN.*memory.offset* = FIELD.*memory.offset*
TYPEDEN.*memory.iTop* = FIELD.*memory.iTop*
FIELD.*memory.sTop* = TYPEDEN.*memory.sTop*
FIELD.*memory.size* = TYPEDEN.*memory.size*
STAT0 → STAT1 ; STAT2 ↦
STAT1.*env.iSt* = STAT0.*env.iSt*
STAT2.*env.iSt* = STAT1.*env.sSt*
STAT0.*env.sSt* = STAT2.*env.sSt*
STAT1.*memory.offset* = STAT0.*memory.offset*
STAT2.*memory.offset* = STAT0.*memory.offset* + STAT1.*memory.size*
STAT1.*memory.iTop* = STAT0.*memory.iTop*
STAT2.*memory.iTop* = STAT1.*memory.sTop*
STAT0.*memory.sTop* = STAT2.*memory.sTop*
STAT0.*memory.size* = STAT1.*memory.size* + STAT2.*memory.size*
STAT1.*code.iLabCtr* = STAT0.*code.iLabCtr*
STAT2.*code.iLabCtr* = STAT1.*code.sLabCtr*
STAT0.*code.sLabCtr* = STAT2.*code.sLabCtr*
STAT0.*code.c* → STAT1.*code.c* ; STAT2.*code.c*
STAT → V := EXP ↦
V.*env.iSt* = STAT.*env.iSt*
EXP.*env.iSt* = V.*env.sSt*
STAT.*env.sSt* = EXP.*env.sSt*
STAT.*memory.sTop* = STAT.*memory.iTop* + 0
STAT.*memory.size* = 0
STAT.*code.sLabCtr* = STAT.*code.iLabCtr*
STAT.*code.c* → assign(V.*code*, EXP.*code*)
V → ID ↦
V.*env.sSt* = V.*env.iSt*
V.*code* → register(1) + const(*getAddress*(*lookup*(V.*env.iSt*, ID)))
V.*recordNmb* = *scopeNmb*(*getType*(*lookup*(V.*env.iSt*, ID)))
V0 → V1.ID ↦
V1.*env.iSt* = V0.*env.iSt*
V0.*env.sSt* = V1.*env.sSt*
V0.*code* → V1.*code* + *getAddress*(*lookupQual*(V0.*env.iSt*,
V1.*recordNmb*, ID))
V0.*recordNmb* = *scopeNmb*(*getType*(*lookupQual*(V0.*env.iSt*,
V1.*recordNmb*, ID)))
EXP → I ↦
EXP.*env.sSt* = EXP.*env.iSt*
EXP.*code* → const(I)
EXP → V ↦
V.*env.iSt* = EXP.*env.iSt*
EXP.*env.sSt* = V.*env.sSt*
EXP.*code* → cont(V.*code*)

$\text{EXP0} \to \text{EXP1}\ op\ \text{EXP2};\ op = +, -, *, / \mapsto$
$\quad \text{EXP1}.env.iSt = \text{EXP0}.env.iSt$
$\quad \text{EXP2}.env.iSt = \text{EXP1}.env.sSt$
$\quad \text{EXP0}.env.sSt = \text{EXP2}.env.sSt$
$\quad \text{EXP0}.code \to \text{EXP1}.code\ op\ \text{EXP2}.code$
$\text{BEXP} \to \text{EXP1}\ relop\ \text{EXP2},\ relop = \langle , \rangle, =, \ldots \mapsto$
$\quad \text{EXP1}.env.iSt = \text{BEXP}.env.iSt$
$\quad \text{EXP2}.env.iSt = \text{EXP1}.env.sSt$
$\quad \text{BEXP}.env.sSt = \text{EXP2}.env.sSt$
$\quad \text{BEXP}.code \to \text{EXP1}.code\ relop\ \text{EXP2}.code$
$\text{STAT0} \to \textbf{if}\ \text{BEXP}\ \textbf{then}\ \text{STAT1}\ \textbf{else}\ \text{STAT2} \mapsto$
$\quad \text{BEXP}.env.iSt = \text{STAT0}.env.iSt$
$\quad E.iSt = \text{BEXP}.env.sSt$
$\quad \text{STAT0}.env.sSt = E.sSt$
$\quad \text{STAT1}.env.iSt = E.iSt$
$\quad \text{STAT2}.env.iSt = \text{STAT1}.env.sSt$
$\quad E.sSt = \text{STAT2}.env.sSt$
$\quad \text{STAT0}.code.c \to \text{JifF}(\text{BEXP}.code, \text{const}(\text{STAT0}.code.iLabCtr));$
$\qquad \text{STAT1}.code.c;$
$\qquad \text{Jmp}(\text{const}(\text{STAT0}.code.iLabCtr + 1));$
$\qquad \text{label}(\text{STAT0}.code.iLabCtr);$
$\qquad \text{STAT2}.code.c;$
$\qquad \text{label}(\text{STAT}.code.iLabCtr + 1)$
$\quad \text{STAT1}.code.iLabCtr = \text{STAT0}.code.iLabCtr + 2$
$\quad \text{STAT2}.code.iLabCtr = \text{STAT1}.code.sLabCtr$
$\quad \text{STAT0}.code.sLabCtr = \text{STAT2}.code.sLabCtr$

References

[1] J. Uhl, S. Drossopoulou, G. Persch, G. Goos, M. Dausmann, G. Winterstein and W. Kirchgässner, An attribute grammar for the semantic analysis of Ada, Lecture Notes in Computer Science **139** (Springer, Berlin, 1982).

[2] J.A. Goguen, J.W. Thatcher and E.G. Wagner, An initial algebra approach to the specification, correctness, and implementation of abstract types, in: R.T. Yeh, Ed., *Current Trends in Programming Methodology, IV: Data Structuring* (Prentice-Hall, Englewood Cliffs, NJ, 1978) 80–149.

[3] J.W. Thatcher, E.G. Wagner and J.B. Wright, Data type specification: Parameterization and the power of specification techniques, *Proc. SIGACT 10th Annual Symposium on Theory of Computing* (1978) 119–132.

[4] J.W. Thatcher, E.G. Wagner and J.B. Wright, More on advice on structuring compilers and proving them correct, *Proc. ICALP 1979*, Lecture Notes in Computer Science **71** (Springer, Berlin, 1979).

[5] H. Ehrig, H.-J. Kreowski, J.W. Thatcher, E.G. Wagner and J.B. Wright, Parameter passing in algebraic specification languages, *Proc. ICALP 1980*, Lecture Notes in Computer Science **85** (Springer, Berlin, 1980).

[6] R.M. Burstall and J.A. Goguen, The semantics of CLEAR, a specification language. Version of Feb. 80, *Proc. 1979 Copenhagen Winter School in Abstract Software Specifications.*

[7] M. Broy, and M. Wirsing, Algebraic definition of a functional programming language and its semantic models, Technische Universität München, Report TUM-I8008 (1980).

[8] H. Christiansen and N. Jones, Control flow treatment in a simple semantics-directed compiler generator, in: D. Bjørner, Ed., *Formal Description of Programming Concepts II* (North-Holland, Amsterdam, 1983) 73–96.
[9] L.M. Chirica and D.F. Martin, An algebraic formulation of Knuthian semantics, *Proc. 17th IEEE Symposium on FOCS* (1977) 127–136.
[10] J. McCarthy and J. Painter, Correctness of a compiler for arithmetic expressions, *Math. Aspects of Comput. Sci., Proc. Symp. Appl. Math.* **19** (1967) 33–41.
[11] H.-D. Ehrich, On the theory of specification, implementation, and parameterization of parameterized data types, *J. ACM* **29** (1) (1982) 206–227.
[12] H. Ehrig, and H.-J. Kreowski, Parameter passing commutes with implementation of parameterized data types, *Proc. 9th ICALP*, Lecture Notes in Computer Science **140** (Springer, Berlin, 1982) 197–211.
[13] M. Ganapathi, Retargetable code generation and optimization using attribute grammars, CSTR-406, Computer Science Department, University of Wisconsin, Madison (1980).
[14] H. Ganzinger, Parameterized specifications: parameter passing and implementation with respect to observability, *Trans. Progr. Languages and Systems* **5**(2) (1983) 318–354.
[15] H. Ganzinger, R. Giegerich, U. Möncke, and R. Wilhelm, A truly generative semantics-directed compiler generator, *ACM Symposium on Compiler Construction*, Boston, *SIGPLAN-Notices* (1982).
[16] M.-C. Gaudel, Correctness proof of programming language translation, in: D. Bjørner, Ed., *Formal Description of Programming Concepts II* (North-Holland, Amsterdam, 1983) 25–42.
[17] J. Guttag, W. Horowitz and D. Musser, Abstract data types and software validation. *Comm. ACM* **21** (12) (1978) 1043–1064.
[18] R. S. Glanville and S.L. Graham, A new method for compiler code generation, *Proc. 5th ACM Symposium on POPL* (1978).
[19] J.A. Goguen, Some design principles and theory for OBJ-0, *Proc. International Conference on Mathematical Studies of Information Processing*, Kyoto (1978).
[20] J.A.Goguen and K. Parsaye-Ghomi, Algebraic denotational semantics using parameterized abstract modules, Lecture Notes in Computer Science **107** (Springer, Berlin, 1981) 292–309.
[21] C.A.R. Hoare, Proof of correctness of data representations, *Acta Informat.* **1** (1972) 271–281.
[22] D.E. Knuth, Semantics of context-free languages, *Math. Systems Theory* **2** (1968) 127–145.
[23] U. Lipeck, An algebraic calculus for structured design of data abstractions (in German), PhD-Thesis, Universität Dortmund (1982).
[24] F.L. Morris, Advice on structuring compilers and proving them correct, *Proc. POPL*, Boston (1973) 144–152.
[25] P. Mosses, A constructive approach to compiler correctness, Lecture Notes in Computer Science **94** (Springer, Berlin, 1980).
[26] P. Mosses, Abstract semantic algebras!, in: D. Bjørner, Ed., *Formal Description of Programming Concepts II* (North-Holland, Amsterdam, 1983) 45–70.
[27] K. Räihä, M. Saarinen, E. Soisalon-Soininen, and M. Tienari, The compiler writing system HLP, Report A-1978-2, Department of Computer Science, University of Helsinki (1978).
[28] K. Ripken, Formale Beschreibung von Maschinen, Implementierungen und optimierender Maschinencodeerzeugung aus attributierten Programmgraphen, Report TUM-I7731, TU München (1977).
[29] J.-C. Raoult and R. Sethi, On metalanguages for a compiler generator, *Proc. ICALP 1982*, Aarhus.
[30] M. Wand, Semantics-directed machine architecture, *Proc. POPL* (1982).
[31] S.N. Zilles, An introduction to data algebras, Working draft paper, IBM Research, San Jose (1975).