Generator Induction in Order Sorted Algebras

Olaf Owe and Ole-Johan Dahl
Institute of Informatics
University of Oslo
Norway

February 1989
(Revised May 1990)

Abstract

Linguistic and semantic consequences of combining the ideas of order sorted algebras (as in OBJ) and generator induction (as in LARCH) are investigated. It is found that one can gain the advantages of both, in addition to increased flexibility in defining signatures and generator bases. Our treatment also gives rise to typing control stronger in a certain sense than that of OBJ, as well as the detection of inherently inconsistent signatures.

Keywords and phrases: Algebraic specification, order sorted algebras, generator induction, functional programming.

Contents

1 Introduction

Goguen et al [5, 6, 7, 8] have introduced the concept of *order sorted algebras* as a basic mechanism in the specification language OBJ. An order sorted algebra is a many-sorted algebra with a partial order defined on the set of sorts, representing the subsort relation. The purpose is to obtain increased flexibility within a regime of strict typing, and to provide a way of dealing with a class of partial functions. Axioms are arbitrary quantifier-free equations over a given signature, and the language semantics is based on an initial algebra assumption, implemented through term rewriting after Knuth-Bendix-like completion.

The technique of *generator inductive function definition* was introduced by Guttag et al [10, 11] and is used in the specification language LARCH and other languages. Guttag axioms have the important properties of preserving consistency as well as "sufficient completeness". They form convergent sets of rewrite rules. In the discussion we make use of elements of a language for specification and programming called ABEL, developed at the University of Oslo [3, 4]. An important part of the language is based on the technique of generator inductive function definition.

The paper is organized as follows: In section 2 we focus on the strength of the OBJ typing mechanism. The type analysis of OBJ, without coercions, is sound in the sense that (i) a well-formed ground term of type (sort) T has a well-defined value in the set associated with T, and (ii) any ground instance of a well-formed term of type T is well-formed and of type T. (A term of type T is also of type T' if T is a subsort of T'). In order to obtain a strong type analysis, one may wish that the reverse of (i) and (ii) hold, i.e. (iii) if every ground instance of a term has a well-defined value in T, then the term is well-formed and of type T, and (iv) if every ground instance of a term is well-formed and of type T, then the term itself is well-formed and of type T. We shall develop requirements that ensure the latter property, called *optimal typing*. In OBJ optimal typing is not possible for terms with multiple variable occurrences, such as $x * x$, which has optimal type natural for integer x. It is clear that (iii) can not be achieved, for instance it is not possible to see statically (without equations) that $x - (x/2)$ is a non-zero natural number for all non-zero natural x.

Terms can sometimes be made well-formed by insertion of coercion functions. For instance, the term $sqrt(NAT(x - y))$ is not well-formed without the NAT-application coercing an integer to a natural. But soundness (part i) is lost if coercion, say from T to a subtype T', is interpreted as undefined outside T'; for instance, the term above is undefined if $y > x$. (Algebraically, a coercion function can be defined as an unspecified total function [9]; an undefined term will then be represented by an irreducible term containing coercion.) Insertion of unnecessary coercions may be avoided with a strong type analysis (for instance, if x is natural and y is negative in the above example).

In chapter 3, after giving an overview of the basic mechanisms of generator induction, we discuss the problem of ensuring ground completeness, by suggesting ways of defining equality constructively.

In the rest of the paper we investigate the linguistic interaction between the concepts of order sorted algebras and generator induction. It is possible to obtain stronger and more

flexible type control than in OBJ. In order to obtain optimal type analysis with OBJ, one must provide a signature with many profiles for each function. A practical problem is that it is difficult to see if there are enough profiles. Another problem is to see whether such a signature satisfies the minimal requirements to monotonicity and regularity. Furthermore the union of two signatures may not satisfy monotonicity, regularity, or optimality, even if both do so separately. It is possible to overcome these problems in the case where each value belongs to a minimal subtype and where the minimal subtypes are mutually disjoint — which is the case for constructively defined (sub)type families. We shall develop methods that from an arbitrary signature compute another satisfying monotonicity, regularity, optimal typing and with the same interpretations as the given one. Optimal typing is possible by systematically rewriting terms with multiple variable occurrences.

As additional advantages, we can detect inherently inconsistent signatures (those that have no interpretation); and we can detect inherently undefined terms such as $x/0$, and $sqrt(-x*x)$ for non-zero x. Coercion (as in $sqrt(NAT(-x*x))$) is here of little help since it would never succeed. Such detections are not possible in OBJ, because emptiness of intersection of domains is not statically known. On the other hand, the undefinedness of terms such as $1/(x-x)$ depends on equations and can not be detected from a signature.

2 Order Sorted Algebras

Let $\mathcal{T}$ be a given finite set of sorts, or *types* as in the terminology of programming languages, and let $\preceq$ be a given partial order on $\mathcal{T}$, called the subtype relation. Each type represents a nonempty set of "values", and the subtype relation represents the inclusion relation on the corresponding value sets. In the following the letters T and D, possibly decorated, stand for types and type products (possibly empty), respectively. A type product over $\mathcal{T}$ is an element of $\mathcal{T}^*$; it represents the corresponding Cartesian product of value sets.

If $T_1 \preceq T_2$, we say that T_1 is a *subtype of* or *included in* T_2, and that T_2 is an *ancestor* of T_1. Two types are said to be *related* if they have a common ancestor. We assume in the following that the subtype relation is such that any two related types have a *unique least* common ancestor. These concepts carry over to type products in the following way: $D_1 \preceq D_2$ holds iff D_1 and D_2 have the same length and the subtype relation holds for each pair of components. D_1 and D_2 are related iff they have the same length and the components are pairwise related. It is reasonable to assume, as in OBJ, that relatedness is a transitive relation on $\mathcal{T}$ (and $\mathcal{T}^*$).

A *signature* is a finite set of *function profiles* of the form $f : D \rightarrow T$, where f is a function symbol. The profile is called a f-profile; it represents a function f total from the domain D into the codomain T. If the former is an empty type product the function is a constant. A signature may contain more than one profile with the same function symbol; they are said to be *coincident*. In order to avoid complications of function overloading we assume in the sequel that the domains and codomains of any two coincident function profiles are *related*. We interpret coincident profiles as representing a single function which is total on each domain, but undefined elsewhere and thus in general partial on any common ancestor of these domains.

In OBJ the following restrictions apply to any signature Σ:

1. *Monotonicity*: Any pair of coincident function profiles with domains D_1, D_2 and corresponding codomains T_1, T_2 must satisfy $D_1 \preceq D_2 \Rightarrow T_1 \preceq T_2$.
2. *Regularity*: For any domain $D \epsilon \mathcal{T}^*$ and function symbol f, the set

$$\{D' \mid D \preceq D' \wedge \exists T \mid f : D' \rightarrow T \ \epsilon \ \Sigma\}$$

must have a unique minimal element if nonempty.

The expression language of an order sorted algebra is the set of *well-formed expressions*, each of which has an associated *minimal type*. Let Σ be a signature and $\mathcal{V}$ a set of typed variables. A well-formed expression of minimal type T over Σ and $\mathcal{V}$ is either

- a variable in $\mathcal{V}$ of type T, or
- a function application $f(e_1, e_2, \ldots, e_n)$, $n \geq 0$, (possibly in infix or "mixfix" notation) where each e_i is a well-formed expression of minimal type T_i $(i = 1..n)$, and there is a f-profile in Σ whose domain is an ancestor of the type product $T_1 \times T_2 \times \ldots \times T_n$, and T is the minimal codomain of such f-profiles.

The definition is meaningful if the signature Σ satisfies the above restrictions. But the regularity restriction is unnecessarily strong: In order to determine the minimal type of an application of f it may not be necessary to identify a unique profile for f, if only the minimal codomain of feasible profiles is unique. Thus, the following may replace the restriction 2 above:

2'. *Weak regularity*: For any domain $D \epsilon \mathcal{T}^*$ and function symbol f, the set

$$\{T \mid \exists D' \mid D \preceq D' \wedge f : D' \rightarrow T \ \epsilon \ \Sigma\}$$

must have a unique minimal element if nonempty.

(This concept has been introduced by Goguen under the term "preregularity". See e.g. [9].) In the sequel the words "expression" and "term" are used interchangeably to mean expression over a signature and variable set determined by the context. Expressions are assumed to be well-formed unless the context indicates otherwise.

The importance of the syntactic type checking embedded in the concept of well-formedness lies in the following semantic invariant: In any model satisfying a given signature and set of equations, a ground expression (without coercions) is well-defined in the model if it is well-formed. (Its value is an element of the set corresponding to the minimal type of the expression.) Well-definedness also holds for well-formed non-ground expressions, given that each variable ranges over the set associated with its type. The last observation corresponds to the following fairly obvious result for OSA term algebras.

Theorem 1 *For monotonic and weakly regular signature Σ and well-formed expression e any ground instance of e is well-formed and its minimal type is included in that of e.*

In an OSA an *instance* of an expression is obtained by replacing each variable, of type T say, by an expression of a type included in T.

An expression violating typing constraints can be transformed to a well-formed one by inserting calls for "coercion" functions (retracts) converting the types of certain subexpressions to subtypes (provided that for each non-well-formed application there is a profile with domain related to the type of the arguments). Coercion from type T to T' where $T' \preceq T$, is a partial function $c : T \rightarrow T'$, whose value is that of the argument if the latter actually belongs to T', and is otherwise undefined. Thus, the semantic well-definedness property does not hold for expressions containing coercions essential for the well-formedness.

Consider an expression of the form $f(e)$, where (e) (a tuple of zero or more components) has minimal type D_e, and there is no f-profile satisfying the typing constraint. Assume that there is a unique maximal domain D related to D_e such that $f : D \rightarrow T$ is in Σ (for some T). Then the well-formed expression of "maximal well-definedness" is $f(c(e))$, where c is the coercion which computes the conjunction of the coercions for those components of e that are *deficient* in the sense that their types are not contained in the corresponding components of D. The coercion c is a (partial) function from D_e to a type $D|D_e$, obtained from D_e by replacing the types of the deficient components of e by the corresponding types in D. Since $D|D_e \preceq D$ holds, $f(c(e))$ is well-formed and its minimal type is determined in the usual way.

If there is no unique maximal domain D as above, but several profiles $f : D_i \rightarrow T_i$ exist whose domains are maximal relatives of D_e, the optimal coercion c must compute the disjunction of the coercions c_i from D_e to $D_i|D_e$. The minimal type of $f(c(e))$ must be taken to be equal to the least common ancestor of the minimal types of the well-formed expressions $f(c_i(e))$.

Completeness of Signatures

We shall now discuss some properties of the typing mechanism defined for well-formed terms; in particular we focus on conditions ensuring a notion of "optimal typing". The following definitions are needed:

- An expression e is *optimally typed* iff the minimal type of e is the least common ancestor of the minimal types of all ground instances of e.
- A type T is *basic* iff it has no subtypes other than itself; and a basic product is one whose components are basic types.
- A type T is *basically equivalent* to a set of types T_i iff the set of basic types included in T is equal to the union of the basic type sets of the T_i's, and similarly for type products.

- $f[D]$ denotes the minimal type of the expression $f(x_1,..,x_n)$ where the variable tuple $(x_1,\ldots,x_n)$ is of type D. $f[D]$ is defined iff $f(x_1,\ldots,x_n)$ is well-formed.
- A signature Σ is said to be *complete* iff for any f, D and set S of type products related to D the following is true: if D is basically equivalent to S and $f[D']$ is defined for each D' in S, then $f[D]$ is defined and is basically equivalent to the set $\{f[D'] \mid D' \in S\}$.

Notice that completeness is a requirement to both $\mathcal{T}$ and Σ. If $\mathcal{T}$ is such that the least common ancestor of any two related types is basically equivalent to the two types, then completeness may be reformulated as follows (restricting Σ only): For any non-basic domain D, $f[D]$ is the least common ancestor of all $f[D_i]$ where D_i is a basic subtype of D, and $f[D]$ is defined when all $f[D_i]$ are defined.

The following theorems express properties about well-formedness and optimal typing in the context of complete signatures.

Theorem 2 *Let Σ be a monotonic, weakly regular, and complete signature, such that for every basic type T there is a well-formed ground term of type T. Then an expression e is well-formed if all its ground instances are well-formed, provided that no variable of non-basic type occurs more than once in e.*

The proof is by induction on the structure of e, with the following induction hypothesis:

- If all ground instances of e are well-defined, then so is e and its minimal type is basically equivalent to the minimal types of its ground instances.

Variables are well-formed by definition. For every basic type B, there is a well-formed ground instance of minimal type B; therefore the induction hypothesis holds for variables.

For a function application $f(e_1,..,e_n)$, each e_i is well-formed with minimal type T_i by the induction hypothesis, and T_i is basically equivalent to the set of minimal types of the ground instances of e_i. Since no variable of non-basic type occurs more than once in e, the types of the argument instances can be combined in all ways. Consequently the product $T_1 \times \ldots \times T_n$ must be basically equivalent to the set of minimal types of the set of ground instances of $(e_1,..,e_n)$. For each product D^j in this set, $f[D^j]$ is defined. By completeness, $f[D]$ is defined and is basically equivalent to the set of all $f[D^j]$.

Theorem 3 *If in addition (to the assumptions of the previous theorem) the set $\mathcal{T}$ is such that each $T \epsilon \mathcal{T}$ is the least common ancestor of its proper subtypes, if any, then a well-formed expression e is optimally typed if no variable of non-basic type occurs more than once in e.*

The condition on $\mathcal{T}$ serves to exclude any type T redundant in the sense of having a single direct subtype. Notice that this implies that any type T is the least common ancestor of all its basic subtypes. The proof is again by induction on the structure of e; the last observation proves the basis of the induction, and the induction step can be done exactly as above.

3 Generator Induction

The notion of generator induction is based on classifying the functions of a Σ-algebra, say on a single type T, as either basic or defined: $\Sigma = \Sigma_{bas} \cup \Sigma_{def}$, where $\Sigma_{bas} \cap \Sigma_{def} = \emptyset$. The purpose of this classification is to provide a more explicit definition of the carrier of the intended Σ-algebra, i.e. the set of values of type T. Informally the meaning of the identification of the subset Σ_{bas} of Σ, say

$$\Sigma_{bas} \triangleq \{g_i : T^{k_i} \to T \mid i = 1..m\},$$

shall be the assertion that *all values of type T can be expressed using the functions $g_1, \ldots, g_m$ only.* For that reason Σ_{bas} is called a *generator basis* (representing a so called "constructor function set") of T, and the associated set of ground terms is called the *generator universe.*

We may thus take these basic ground terms as *names on the abstract T-values.* If they are in a one to one correspondence with the intended values, Σ_{bas} is said to be a *one-to-one generator basis*, otherwise it is said to be *many-to-one.*

Clearly, the generator universe is partially ordered by the subterm relation; and since that relation is well founded it gives rise to an induction principle which is called generator induction. Therefore, the meaning of a generator basis specification may be formalized by introducing, in an underlying system for first order logic with equality, an inference rule for such induction, *defined as induction over T.*

$$T\text{-induction:} \quad \frac{P[x_1/x], \ldots, P[x_{k_i}/x] \;\vdash P[g_i(x_1, \ldots, x_{k_i})/x];\; i = 1..m}{\vdash \forall x : T \mid P}$$

where P is a formula, $x_1, \ldots, x_{k_i}$ are fresh variables, $P[t/x]$ stands for P with t substituted for x, and the expressions $P[x_j/x]$ are induction hypotheses. Generator induction is useful for function definition as well as theorem proving.

Example.

Let Σ^{Nat} be the signature of an algebra on natural numbers,

$$\Sigma^{Nat} = \{0 : \to Nat,\ \mathbf{S} : Nat \to Nat,\ + : Nat^2 \to Nat,\ \ldots\},$$

where the function symbols are intended to correspond to the concepts of zero, successor, and addition. Defining $\Sigma^{Nat}_{bas} = \{0 : \to Nat,\ \mathbf{S} : Nat \to Nat\}$ leads to the generator universe $\{0,\ \mathbf{S}0,\ \mathbf{SS}0, \ldots\}$, and to an induction principle which corresponds to ordinary mathematical induction.

$$Nat\text{-induction:} \quad \frac{\vdash P[0/x];\quad P[x_1/x] \;\vdash P[\mathbf{S}x_1/x]}{\vdash \forall x : Nat \mid P}$$

In this case the basic ground terms are clearly in a one to one correspondence with the intended abstract values. Notice that for instance the subsignature $\{0 : \to Nat,\ + : Nat^2 \to Nat\}$ would not provide a generator basis for Nat.

The notion of generator basis is easily generalized to the domain of mixed algebras. Let $\mathcal{T}$ be the set of types, mutually disjoint. We assume that there is exactly one generator basis defined for each $T \epsilon \mathcal{T}$, denoted Σ^T_{bas} and consisting of functions with codomain T. Let $\Sigma_{bas} = \bigcup_{T \epsilon \mathcal{T}} \Sigma^T_{bas}$. Then the generator universe for T is defined as the subset of ground expressions over Σ_{bas} which are of type T.

The following syntactic criterion is a necessary and sufficient condition for each type in $\mathcal{T}$ to have a nonempty generator universe (and value set).

- There exists a total order on $\mathcal{T}$ such that the generator basis of each type $T \epsilon \mathcal{T}$ contains a function with domain D, such that each component (if any) of D precedes T in the total order.

We consider a specification system such that the functions of Σ_{def} must be defined constructively in terms of basic functions, by equational axioms possibly using generator induction. A *direct definition* of a function f is an axiom of the form

$$f(\overline{v}) = e$$

where $\overline{v}$ is a list of distinct variables, and e is a quantifier-free expression in these variables, basic functions, and functions previously defined.

Let the generator basis of $T \epsilon \mathcal{T}$ be $\Sigma^T_{bas} = \{g_i : D_i \rightarrow T \mid i = 1..m\}$, and assume that $f : D_x \times T \times D_y \rightarrow T' \; \epsilon \; \Sigma_{def}$, Following Guttag, a definition of f, using *generator induction* with respect to the indicated T-argument, is a set of equations whose left hand sides are obtained from the one above by replacing the inductive argument in the left hand side by each generator function in turn, applied to distinct fresh variables:

$$f(\overline{x}_i, g_i(\overline{z}_i), \overline{y}_i) = e_i, \;\; i = 1..m.$$

Recursive application of functions being defined is permitted, subject to syntactic restrictions strong enough to guarantee termination with respect to term rewriting — specifically, the T-argument of any recursive application must be a proper subterm of that of the left-hand-side. (It follows that direct definitions must be non-recursive.)

Assuming that all functions in Σ_{def} are defined according to the above rules, the set of axioms comprises a *convergent set of rewrite rules.* (Confluence follows from the absence of left hand side superpositions). Also all ground Σ-terms have *basic* normal forms; and all basic ground terms are irreducible. In this sense the *value* of any ground Σ-term can be computed by term rewriting. In algebraic terms it follows that the carrier of the Σ-algebra is given by some Σ_{bas}-algebra. Whether or not the latter is completely specified, depends on how equality relations are treated formally.

In the ABEL language we may collect the set of Guttag axioms for a function into a single function definition, whose right hand side is a generalization of the Pascal case construct (cf. also Standard ML [12]).

$$
\begin{array}{lcll}
f(\overline{x}, z, \overline{y}) & = & \textbf{case } z \textbf{ of} & g_1(\overline{z}_1) \rightarrow e_1 \\
 & & [] & g_2(\overline{z}_2) \rightarrow e_2 \\
 & & & \cdots\cdots\cdots \\
 & & [] & g_n(\overline{z}_n) \rightarrow e_n \\
 & & \textbf{fo} &
\end{array}
$$

Notice that for the purpose of term rewriting a definition whose right hand side contains case-expression(s) corresponds to a set of rewrite rules, and the rule selection is performed by the rewriting algorithm by a pattern matching mechanism deciding the indicated discriminations. The case construct immediately leads to useful generalizations of the Guttag schema, like nested induction and conditional axioms (the latter by discriminating on expressions other than variables).

These generalizations preserve confluency since there will be no superpositions in the left hand sides of the corresponding case-free axioms. The syntactic termination control may be generalized in many ways, more or less powerful. For the purpose of the examples occurring in the sequel it is sufficient to use the lexicographic order induced by a generalized subterm relation, for each function according to a fixed permutation of its arguments. The generalized subterm relation requires one or more subterms to be replaced by proper subterms.

In the sequel we refer to function definitions in this format as *ABEL style* axioms. The traditional if-then-else construct may be defined as a case discriminating on an expression of type *Bool*, but it may be more useful to treat that construct as an ordinary function with respect to term rewriting. The examples show that functions other than primitive recursive ones are definable in ABEL style. Obviously, however, no syntactic termination control can be strong enough to cater for the whole class of general recursive functions.

Example

The Ackermann function on natural numbers may be defined as follows:

$ack(x, y) =$ **case** x **of** $0 \rightarrow \mathbf{S}x$ [] $\mathbf{S}x \rightarrow$ **case** y **of**
$0 \rightarrow ack(x, \mathbf{S}0)$ [] $\mathbf{S}y \rightarrow ack(x, ack(\mathbf{S}x, y))$ **fo fo**

where the redeclarations of x and y hide the outer ones. The definition is equivalent to the following case-free axioms:

$ack(0, y) = \mathbf{S}y$
$ack(\mathbf{S}x, 0) = ack(x, \mathbf{S}0)$
$ack(\mathbf{S}x, \mathbf{S}y) = ack(x, ack(\mathbf{S}x, y))$

All three recursive applications satisfy the indicated syntactic restriction, since the first argument becomes smaller or equal, and in the latter case the second argument becomes strictly smaller (being a proper subterm).

We now consider the formal treatment of equality. Clearly the underlying logical system defines equality as a (polymorphic) congruence relation, by means of axioms or inference rules expressing reflexivity, symmetry, transitivity, as well as substitutivity. If the only

additional axioms are function definitions, equality on ground terms is not in general fully defined, and this corresponds to the intuition that different basic ground terms may well be intended to denote the same abstract value. It is clear, however, that the total axiom set is *consistent*.

Let $Bool \epsilon \mathcal{T}$ and $= : T^2 \rightarrow Bool \ \epsilon \ \Sigma^T_{def}$ for all $T \epsilon \mathcal{T}$. Let also $\Sigma^{Bool}_{bas} = \{false, true\}$, and $\neg(true = false)$ be an axiom. We suggest that equality can be treated as an ordinary defined function for each of the other types, axiomatized constructively by function definitions according to the above rules. If the equalities thus defined are in fact congruence relations, then consistency is preserved. The logic is also *ground complete*, in the sense that all ground formulas, i.e. expressions of type *Bool* including equations, are reduced to *true* or *false* by term rewriting. The equalities induce *equivalence classes* on the generator universe of each type T, which represent the abstract T-values. The family of corresponding quotient sets is (isomorphic to) the carrier of the resulting mixed Σ-algebra.

For a type T with a one-to-one generator basis $\Sigma^T_{bas} = \{g_i : D_i \rightarrow T \mid i = 1..m\}$ the equality relation is defined as follows by (nested) generator induction.

$$(x = y) \ = \ \textbf{case} \ (x, y) \ \textbf{of} \ \underset{i=1..m}{[]} \ (g_i(\overline{z}_i), g_i(\overline{w}_i)) \rightarrow \overline{z}_i = \overline{w}_i \ [] \ \textbf{others} \ \rightarrow false \ \textbf{fo}$$

A specification system could construct this definition for any given generator basis specified as being one-to-one. If all types in $\mathcal{T}$ have one-to-one generator bases, the definitions express syntactic equality of basic ground expressions, which implies that all congruence axioms are necessarily satisfied, and that logical consistency therefore is preserved. In that case the carrier of the Σ-algebra is given by the initial Σ_{bas}-algebra.

The mathematical and conceptual simplicity of one-to-one generator bases indicates that one usually tries to find bases with the one-to-one property. However, that is not always possible (see the example below). If a many-to-one generator basis must be used for a type T, it remains to define the intended equivalence classes on basic ground terms, by explicit definition of abstract T-equality or by other means. In the former case there is a heavy proof burden to show that the defined equality function is in fact a congruence relation. In any case there is a constant danger to lose consistency through the subsequent use of generator induction over T in function definitions. Technically this may happen as the result of violating congruence axioms; intuitively the reason is that generator induction now reveals structure in the generator universe which should be hidden within the equivalence classes.

We may reduce the proof burden by defining a suitable subset of basic ground terms as being *unique representatives* of the equivalence classes. It is sometimes possible to include equational axioms on basic terms which preserve term rewrite convergence, such that the irreducible ground terms are unique representatives, see below. Then a ground complete system can be obtained as in OBJ. However, proving rewrite convergence and logical consistency of the whole axiom set is non-trivial in general.

Another technique is to introduce an explicit function $rep : T \rightarrow T$, defined in ABEL style, for computing the representatives, and defining T-equality as syntactic equality on them.

$$(x = y) \ = \ \textbf{case} \ (rep(x), rep(y)) \ \textbf{of} \ \underset{i=1..m}{[]} \ (g_i(\overline{z}_i), g_i(\overline{w}_i)) \rightarrow \overline{z}_i = \overline{w}_i \ [] \ \textbf{others} \rightarrow false \ \textbf{fo}$$

Then consistency, as well as rewrite convergence will be preserved automatically, provided that all case-discriminants of type T (except those used in the definition of *rep* itself) are of the form $rep(t)$. Notice that the discriminants of the proposed equality definition do have this form. Consistency follows from the fact that only the unique representatives are now considered in inductive definitions of other functions.

This idea of safeguarding generator inductive definitions by applying the *rep*-function to the discriminant, guarantees logical consistency for any choice of *rep*-function. However, any intuitively reasonable *rep*-function will be such that $x = rep(x)$ holds, which means that *rep* should be idempotent with respect to syntactic equality on basic ground terms.

Example

It appears that a type *Set* of elements of a given (infinite) type T has no (finite) one-to-one generator basis. We may, however, define the type through the following many-to-one generator basis:

$$\Sigma_{bas}^{Set} = \{\emptyset :\rightarrow Set,\ add : Set \times T \rightarrow Set\}$$

where an expression $add(s, x)$ is supposed to compute the union of the sets s and $\{x\}$. Assuming that there is a total order $<<$ on T (the subexpression order on any generator universe may easily be extended to a total order) we may define as unique representatives those generator expressions which correspond to repetition-free sorted sequences. We define a function $rep : Set \rightarrow Set$ computing these representatives.

$$\begin{array}{lcl} rep(s) & = & \textbf{case } s \textbf{ of } \emptyset \rightarrow \emptyset \ [] \ add(t,x) \rightarrow \\ & & \quad \textbf{case } rep(t) \textbf{ of } \emptyset \rightarrow s \ [] \ add(u,y) \rightarrow \\ & & \quad\quad \textbf{if } x = y \textbf{ th } add(u,y) \textbf{ el} \\ & & \quad\quad\quad \textbf{if } y << x \textbf{ th } add(add(u,y),x) \textbf{ el} \\ & & \quad\quad\quad\quad add(rep(add(u,x)),y) \textbf{ fi fi fo fo} \end{array}$$

Notice that this function is idempotent. It also has the useful property that any type *Set* subterm of a *Set* representative is itself a representative. *Set* equality can now be defined as above without losing logical consistency. Notice that properties of representatives may be assumed when defining functions by (guarded) generator induction. For instance, we may define the cardinal function as follows, assuming $card : Set \rightarrow Nat \in \Sigma_{def}$, using the fact that no member of a representative is repeatedly added.

$$card(s) \quad = \quad \textbf{case } rep(s) \textbf{ of } \emptyset \rightarrow 0 \ [] \ add(t,x) \rightarrow \mathbf{S}\ card(t) \textbf{ fo}$$

This definition represents the following two conditional axioms.

$$\begin{array}{lcll} card(s) & = & 0, & \text{if } rep(s) \text{ is of the form } \emptyset \\ card(s) & = & \mathbf{S}\ card(t), & \text{if } rep(s) \text{ is of the form } add(t,x) \end{array}$$

An obvious drawback of using function definitions of this form is that the resulting term rewriting system will be quite inefficient (and in this case only applicable to the evaluation of ground terms).

An alternative is to ensure that all constructed values of the type in question are in representative form, by applying the *rep* function to any generator application (except in case-discriminators and in the definition of *rep*). Then, if the *rep* function has the subterm property mentioned above, all allowed terms of this type have representative values, *including variables introduced in left hand sides and discriminators*. Thus no discriminant guards will be necessary, which implies that *rep* itself is simplified. And equality is defined as in the case of one-to-one generator basis. Since the subterm property can be determined syntactically, logical consistency is syntactically guaranteed.

Example continued

It now suffices to define the *rep*-function on each generator as follows:

$$\begin{array}{lcl} rep(\emptyset) & = & \emptyset \\ rep(add(t,x)) & = & \textbf{case } t \textbf{ of } \emptyset \rightarrow add(t,x) \ [] \ add(u,y) \rightarrow \\ & & \quad \textbf{if } x = y \textbf{ th } t \textbf{ el} \\ & & \quad\quad \textbf{if } y << x \textbf{ th } add(t,x) \textbf{ el} \\ & & \quad\quad\quad add(rep(add(u,x)),y) \textbf{ fi fi fo} \end{array}$$

The recursive call satisfies our syntactic restrictions since u is a proper subterm of $add(u,y)$ which again is a subterm of $add(t,x)$. Since we may assume that t is a representative, this definition is more efficient than the previous one.

It should be legal to shorthand those applications of *rep* which can be proved redundant. For instance, the delete-function may be defined as follows:

$$\begin{array}{lcl} del(s,x) & = & \textbf{case } s \textbf{ of } \emptyset \rightarrow \emptyset \ [] \ add(t,y) \rightarrow \\ & & \quad \textbf{if } y = x \textbf{ th } t \textbf{ el } add(del(t,x),y) \textbf{ fi fo} \end{array}$$

The occurrence of $\emptyset$ in the first branch is obviously allowed; for the last occurrence of add it must be proved that $add(del(t,x),y)$ evaluates to a representative.

A natural OBJ-like technique might be to include equational axioms on basic terms expressing commutativity and idempotence:

$$\begin{array}{lcl} add(add(s,x),y) & = & add(add(s,y),x), \\ add(add(s,x),x) & = & add(s,x), \end{array}$$

Used as rewrite rules (the commutativity rule guarded by the condition $y << x$), they lead to exactly the same representatives as above. But proving rewrite convergence and consistency is now non-trivial, even if the non-basic functions are axiomatized in ABEL style. (Actually, equality defined as above destroyes confluency.)

It is clear that the above rigid rules of axiomatization in some ways restrict the class of definable functions. In particular, only total functions are catered for. However, a certain class of partial functions may be covered by introducing undefined constants ("error symbols"), and/or by OSA techniques as explained in the next chapter.

Function definition by generator induction can be seen as a kind of functional programming on data structures specified by the generator basis. For that reason we believe that it is a specification tool easier to use for trained programmers than arbitrary equational axioms

are likely to be. The gain in conceptual simplicity is clearly greatest whenever one-to-one generator bases can be used. The following are more specific advantages:

- Axioms have a predefined form which guides the user.
- Consistency is established by syntactic checks (at least for one-to-one generator bases).
- The same is true for ground completeness in the sense of logic.
- The axioms form a convergent set of rewrite rules. (Termination follows from syntactic restrictions on the use of recursion.)
- There is an induction principle for every infinite type usable for proof purposes.
- Proofs by Knuth-Bendix completion are simplified if all generator bases are one-to-one. (The test for inductive reducibility becomes trivial.)

4 Order Sorted Generator Induction

It is possible to combine the idea of order sorted algebras and that of generator induction so as to gain the advantages of both. As before, a signature Σ is partitioned into Σ_{bas} and Σ_{def}. Coincident function profiles are permitted in all of Σ, but no two coincident profiles may both belong to Σ_{bas}.

Consider a type T not included in any other type, and with generator basis Σ^T_{bas} in which the set of types occurring as codomains is exactly the set of basic types included in T (if there is only one, T itself must be basic). Then we know that the union of the value sets of these basic types is that of T itself. Furthermore, if the generator basis is one-to-one the basic types are necessarily *disjoint*. Also for a many-to-one generator basis it is natural to require that the basic types be disjoint; in that case each basic type may have to be the codomain of more than one generator, and disjointness may require a semantic proof.

As a result of these restrictions it becomes possible to syntactically compute type unions and intersections as well as the subtype relation, simply by representing any type as the set of (names of) its basic subtypes. Thus, an empty set represents an empty type, and the operations $\cup, \cap, \preceq$ on types are defined as respectively $\cup, \cap, \subseteq$ on the representations. Each type is interpreted by the union of the value-sets of its basic subtypes. We no longer allow coercion from one subtype to another if the two have an empty intersection.

Let T be as above. The set of types consisting of the given basic subtypes of T and all unions of two or more of them, including T itself, is called a *type family headed by* T, and *based on* the basic types. Adding an empty type the family becomes a lattice with respect to the operations of union and intersection. Let the total set $\mathcal{T}$ of types consist of type families, and let the subtype relation on $\mathcal{T}$ be the disjunction of the subtype relations on the individual families. If the families in $\mathcal{T}$ are mutually disjoint (except for the trivial empty type) then relatedness as defined in section 2 is a transitive relation.

Example

We may define a type family of integer types headed by *Int* and based on *Nat1*, *Neg1*, and *Zero*, representing the positive and negative integers and the singleton set of zero, respectively. The other members of the family are:

$$\begin{array}{lcl} Nat & = & Nat1 \cup Zero, \\ Neg & = & Neg1 \cup Zero, \quad \text{and} \\ Nzero & = & Nat1 \cup Neg1. \end{array}$$

A one-to-one generator basis Σ_{bas}^{Int} may consist of:

$$\begin{array}{lll} 0: & \rightarrow Zero & \\ \mathbf{S}\hat{}: & Nat \rightarrow Nat1 & \text{(successor)} \\ \mathbf{N}\hat{}: & Nat1 \rightarrow Neg1 & \text{(negation)} \end{array}$$

where boldface script is used to represent operator symbols, and the sign $\hat{}$ identifies an argument position. Notice that the one-to-one property is obtained by the trick of restricting the domains of the generators $\mathbf{S}$ and $\mathbf{N}$ to suitable subtypes of *Int*. An alternative one-to-one generator basis would result by replacing the negation profile by one for the predecessor function restricted to non-positive integers $\mathbf{P}\hat{}: Neg \rightarrow Neg1$. It would be useful to extend these functions to apply to arguments from all of *Int*. We may do so by introducing the following profiles in Σ_{def}:

$$\begin{array}{ll} \mathbf{S}\hat{}: & Neg1 \rightarrow Neg \\ \mathbf{N}\hat{}: & Neg1 \rightarrow Nat1 \\ \mathbf{N}\hat{}: & Zero \rightarrow Zero \end{array}$$

(and possibly $\mathbf{P}\hat{}: Nat1 \rightarrow Nat$).

Let t, u, and v be terms of type *Nat*, *Neg1*, and *Int*, respectively. Then $\mathbf{S}t$, $\mathbf{S}u$, and $\mathbf{S}v$ are applications of the basic part of S, the non-basic part, and the whole of $\mathbf{S}$, respectively. The two first expressions are respectively of type *Nat1* and *Neg*. The type of the last one should clearly be *Int*; so the profile $\mathbf{S}\hat{}: Int \rightarrow Int$, obtained by taking the "union" of the two given $\mathbf{S}$-profiles, should be added to Σ_{def} (and similarly for the other functions).

Axioms defining the non-basic parts of these functions semantically are given in chapter 6.

We restrict type union and type intersection to only apply to related types. Then $\mathcal{T}$ itself is a lattice with respect to these operations. We now extend the set $\mathcal{T}^*$ of domains to a set $\mathcal{D}$ which includes all unions of related Cartesian products. The elements of $\mathcal{D}$ can be seen as unions of products of basic types, unique up to union reordering. This is a result of the fact that Cartesian multiplication distributes over set union. The fact that basic types are disjoint implies that the intersection of any pair of (related) elements of $\mathcal{D}$ is again a union, possibly empty, of basic products. Thus, $\mathcal{D}$ too is a lattice. The subtype relation $\preceq$ is extended to $\mathcal{D}$ meaning value set inclusion. Let $\mathcal{T}_{bas}$ be the set of basic types in $\mathcal{T}$. Then all relevant operators on $\mathcal{D}$ are easily defined syntactically by representing each element of $\mathcal{D}$ by the appropriate subset of $(\mathcal{T}_{bas})^*$.

The union and intersection of two coincident function profiles, $f : D_1 \to T_1$ and $f : D_2 \to T_2$, are defined as $f : D_1 \cup D_2 \to T_1 \cup T_2$ and $f : D_1 \cap D_2 \to T_1 \cap T_2$, respectively. If $D_1 \cap D_2$ is empty the intersection is said to be empty. An interpretation of two f-profiles must also be an interpretation of their union and nonempty intersection, since the two former f-profiles represent a function total from $D_1 \cup D_2$ into $T_1 \cup T_2$ such that its restriction to D_1 is into T_1 and its restriction to D_2 is into T_2. Consequently its restriction to $D_1 \cap D_2$, if nonempty, must be into $T_1 \cap T_2$, which is only possible when $T_1 \cap T_2$ is also nonempty. Otherwise, the function cannot be represented and the two profiles are inconsistent. A signature is said to be (syntactically) *consistent* if it has an interpretation with non-empty interpretation of every subtype.

The check on Σ_{bas}, given in section 3, providing a guarantee that all types thereby defined have nonempty value sets, can easily be adjusted to our context of order sorted generator induction:

- There exists a total order on Σ_{bas} such that the domain of each basic function profile is a product (possibly empty) of types, where each type includes the codomain of a basic profile occurring earlier in the order.

This implies that for every basic type T there is a well-formed basic ground expression of type T.

Examples

Given the following set of profiles for integer division:

$$\begin{array}{ll} \hat{}/\hat{}: & Nat \times Nat1 \to Nat \\ & Neg \times Neg1 \to Nat \\ & Nat \times Neg1 \to Neg \\ & Neg \times Nat1 \to Neg \end{array}$$

By means of union and intersection we may generate these additional profiles with domains in $\mathcal{T}^*$:

$$\begin{array}{ll} \hat{}/\hat{}: & Int \times Nzero \to Int \\ & Zero \times Nzero \to Zero \end{array}$$

Also from the following first set of profiles for integer multiplication the second one follows, and vice versa:

$$\begin{array}{ll} \hat{} * \hat{}: & Nat \times Nat \to Nat \\ & Neg \times Neg \to Nat \\ & Nat \times Neg \to Neg \\ & Neg \times Nat \to Neg \\ & Nzero \times Nzero \to Nzero \end{array}$$

$$\begin{array}{ll} \hat{} * \hat{}: & Int \times Int \to Int \\ & Nat1 \times Nat1 \to Nat1 \\ & Neg1 \times Neg1 \to Nat1 \\ & Nat1 \times Neg1 \to Neg1 \\ & Neg1 \times Nat1 \to Neg1 \\ & Zero \times Int \to Zero \\ & Int \times Zero \to Zero \end{array}$$

A profile $f : D \to T$ is said to be *covered by* a coincident one $f : D' \to T'$ iff $D \preceq D'$ and $T' \preceq T$. It is said to be *redundant by* a signature Σ iff covered by a profile in Σ, and

redundant in Σ (or simply "redundant" if the identity of Σ is clear from the context) iff covered by some *other* profile in Σ. Redundant profiles do not contribute information, and may be removed from a signature without changing its interpretations.

A unique, monotonic, regular and complete signature $\bar{\Sigma}$ can be obtained from an arbitrary signature Σ by first removing all redundant function profiles and then taking the closure with respect to non-redundant union as well as non-redundant, nonempty intersection of profiles. It is quite obvious that $\bar{\Sigma}$ has exactly the same interpretations as Σ. In particular Σ is consistent if and only if $\bar{\Sigma}$ contains no function profile with empty codomain.

The following lemma states that $\bar{\Sigma}$ is well-defined, and the theorems that it has the desired properties:

Lemma 1 *The process of extending a set of non-redundant function profiles by repeatedly adding non-redundant profiles formed by either union or nonempty intersection, is convergent.*

Termination follows from the fact that there are only finitely many profiles related to those in Σ. Confluence follows from the fact that union distributes over intersection and vice versa.

Theorem 4 *For arbitrary consistent* Σ, $\bar{\Sigma}$ *is monotonic and regular.*

Obviously, if $\bar{\Sigma}$ were not monotonic, it would contain a redundant profile. For proving regularity consider, for given D and f, the intersection, say $f : D' \to T'$, of all f-profiles $f : D_i \to T_i$ in $\bar{\Sigma}$ such that $D \preceq D_i$ (we may assume that this set is nonempty). The intersection must satisfy $D \preceq D'$ and $T' \preceq T_i$ for each i, and it must occur in $\bar{\Sigma}$ unless it is redundant, in which case $\bar{\Sigma}$ contains $f : D'' \to T''$, where $T'' \preceq T'$ and $D' \preceq D''$. Since $D \preceq D''$ follows, T'' must equal T_j for some j, and must also equal T'. Since $T' = T_j \preceq T_i$ for each i, the set $\{T_1, T_2, \ldots\}$ has a unique minimal element, and weak regularity follows. Strong regularity is then a consequence of the following lemma:

Lemma 2 *For a given function symbol,* $\bar{\Sigma}$ *has at most one profile with a given codomain, and at most one profile with a given domain.*

If there were two profiles with the same domain their intersection would make both redundant. If there were two profiles with the same codomain their union would make both redundant.

Consequently, well-formedness and minimal type of expressions may be decided in terms of $\bar{\Sigma}$ as in the order sorted case, cf. section 2; the only requirement to the given signature Σ is consistency. Coercion, however, would be simpler than explained in section 2, avoiding the most difficult case. This follows from the following lemma:

Lemma 3 *For a given function symbol* f *there is a unique profile in* $\bar{\Sigma}$ *with a maximal domain.*

If there were several maximal profiles, their union would have a larger domain.

The following lemmas state that $\bar{\Sigma}$ is complete and minimal in a certain sense.

Theorem 5 *For arbitrary consistent* Σ, $\bar{\Sigma}$ *is complete.*

Completeness is quite obvious, since $\mathcal{T}$ is closed with respect to union and since $\bar{\Sigma}$ is closed with respect to non-redundant union.

Theorem 6 *The set* $\bar{\Sigma}$ *is minimal in the sense that no function profile can be removed without loosing either interpretation equivalence with* Σ*, regularity, or completeness.*

This implies that $\bar{\Sigma}$ is also minimal in the sense that no profile can be removed without weakening the concept of minimal type as defined in section 2. Furthermore, if $\bar{\Sigma}$ is consistent and satisfies the syntactic requirement above to the total order of Σ_{bas}, then all assumptions of theorem 3 (on the signature) are satisfied; consequently, an expression e in which no variable occurs more than once, is well-formed iff all its ground instances are well-formed, and its minimal type is the least common ancestor of the minimal types of all its ground instances.

Furthermore, a term t with more than one occurrence of a variable x of non-basic type, say T, may be rewritten as

$$\textbf{case } x \textbf{ of } \mathop{[]}_{i=1..k_T} T_i \rightarrow t \textbf{ fo}$$

where $T_1, \ldots, T_{k_T}$ are the basic subtypes of T. With respect to the typing algorithm the case-construct with a discriminant of minimal type T behaves as a polymorphic function

$$\textbf{case}\hat{} \ldots \hat{}\,\textbf{fo} : T \times U_1 \times \ldots \times U_{k_T} \rightarrow U$$

where $U_1, \ldots, U_{k_T}$ are the minimal types of the alternatives, and U is their least common ancestor. (Σ_{def} may be assumed to contain one such profile for every non-basic type T and every list of k_T mutually related types U_i.) Notice that x is of a basic type in each branch, and therefore the multiple occurrences do not violate optimal typing. We may thus obtain optimal typing with the original type analysis for any well-formed expression t, by systematically rewriting t in this way for all variables of non-basic type occurring more than once. For instance, the case-rewriting makes *Nat* the minimal type of $x * x$, for $x : Int$, using the profiles for the multiplication operator occurring in a previous example.

The theorems and lemmas above do not depend on the existence of a generator basis, they only assume the existence of disjoint basic (sub-)types. The fact that the closure $\bar{\Sigma}$ of any signature Σ in this case can be constructed mechanically and has the same interpretations as Σ, and that only consistency is required of Σ leads to flexibility of specification. It becomes possible to strengthen a signature Σ by adding new profiles thus providing additional syntactic information with no need of rewriting any part of Σ. Also signatures with overlapping sets of types and functions can be joined together, provided that the respective subtype and type disjointness relations are not in conflict. In particular, any ordinary order

sorted signature may be combined with (and checked against) one whose types are defined by order sorted generator induction. Notice that the strengthening of a signature preserves well-formedness of expressions.

Cunningham et al [2] have also defined a signature closure by lifting coincident profiles; however, this closure guarantees neither completeness, regularity nor monotonicity (and does not preserve interpretations in the general case of OSA).

5 Implementation Considerations

For implementing the computation of minimal types of expressions in an efficient way it is useful to extend the subtype relation on $\mathcal{T}$ to a total order. That is easy, based on the representation of each type as a union of basic types, given some total order on the basic types in $\mathcal{T}$. For types such that neither is a subtype of the other, the smallest may be defined as the one with the smallest basic type not contained in the other. Now, by lemma 2, each maximal set of coincident function profiles in $\bar{\Sigma}$ is totally ordered by the order on codomains. Consider an expression $f(e)$, where (e) is tuple of minimal type D_e. It follows from the regularity of $\bar{\Sigma}$ that the minimal type of $f(e)$ is determined by searching the list of f-profiles in the order of increasing codomains, looking for the first domain D such that $D_e \preceq D$. (If none occurs the expression is not well-formed.)

The subtype test on domains, used in the algorithm, can be implemented by representing each domain D canonically as indicated in the last section, i.e. as $\tilde{D} \subseteq (\mathcal{T}_{bas})^*$. Then $D_1 \preceq D_2$ iff $\tilde{D_1} \subseteq \tilde{D_2}$.

This implementation, although theoretically simple, is not easy to realize in an efficient manner; and the canonic representation is also impracticable as a linguistic device:

- Lists of basic products tend to be long and unreadable.
- The syntactic properties of functions are usually expressible in a natural way by a few profiles with domains in $\mathcal{T}^*$.
- The minimal type of an argument list is directly expressed as a type product in $\mathcal{T}^*$.

Fortunately, for any signature $\bar{\Sigma}$ obtained as above (with domains in $\mathcal{D}$) it is easy to construct a signature $\hat{\Sigma}$ equivalent with respect to interpretation, whose profiles have domains in $\mathcal{T}^*$, and is monotonic, weakly regular, complete, and of minimal cardinality. The construction may be in two steps, starting with $\hat{\Sigma} = \emptyset$:

1. For each profile $f : DD \rightarrow T$ in $\bar{\Sigma}$ add to $\hat{\Sigma}$ the set $\{f : D \rightarrow T \mid D \preceq DD,\ D \epsilon \mathcal{T}^*\}$.
2. Delete from $\hat{\Sigma}$ all redundant profiles.

Let Σ_1 be the result of step 1. It is fairly clear that Σ_1 is interpretation equivalent to $\hat{\Sigma}$, as well as regular and complete (with respect to domains in $\mathcal{T}^*$, not $\mathcal{D}$). This follows from the fact that the subset of Σ_1 corresponding to any profile $f : DD \rightarrow T$ in $\bar{\Sigma}$ has elements corresponding to *all* $\mathcal{T}^*$ domains included in DD (including those occurring in the canonical representation of DD).

It is clear that step two preserves interpretation equivalence and completeness, and that monotonicity and minimality (in some sense) is attained. Regularity may be lost, but $\hat{\Sigma}$ is at least weakly regular. This follows by reasoning as in the proof of Theorem 4, since any nonempty intersection of profiles in $\hat{\Sigma}$ is either contained in $\hat{\Sigma}$ or is redundant by $\hat{\Sigma}$.

It is possible to construct $\hat{\Sigma}$ directly from a user defined signature Σ (with domains in $\mathcal{T}^*$). The construction is based on the syntactic concept of a *merge* of two related domains, $D = T_1 \times \ldots \times T_n$ and $D' = T'_1 \times \ldots \times T'_n$:

$$merge(D, D') \stackrel{\text{def}}{=} \{T_1 \cap T'_1 \times \ldots \times T_{k-1} \cap T'_{k-1} \times T_k \cup T'_k \times T_{k+1} \cap T'_{k+1} \times \ldots \times T_n \cap T'_n \mid k = 1..n\}$$

(Notice the union operator in the k'th component.) We also define a merge operation on coincident profiles:

$$merge(\sigma, \sigma') \stackrel{\text{def}}{=} \{f : D'' \rightarrow T \cup T' \mid D'' \epsilon\ merge(D, D')\}$$

where σ is $f : D \rightarrow T$ and σ' is $f : D' \rightarrow T'$.

It is easy to prove that each profile covered by $\sigma \cup \sigma'$, and with domain in $\mathcal{T}^*$, is redundant by the set $merge(\sigma, \sigma') \cup \{\sigma, \sigma'\}$, and vice versa. This shows that $\hat{\Sigma}$ can be constructed in the following two steps:

1. Construct the closure of Σ with respect to merging and intersection.
2. Remove redundant profiles.

Notice that merging and redundancy removal do not in general commute. The set of profiles with domains in $\mathcal{T}^*$ is closed with respect to profile merging and non-empty intersection.

$\hat{\Sigma}$, as well as Σ itself, are easily represented in terms of types in $\mathcal{T}$. Notice that $\mathcal{T}$ is closed with respect to union and intersection of related types. The algorithm for computing minimal types of expressions is now easy to implement efficiently; the same technique of ordering profiles is useful (those with the same codomain, if any, may be ordered arbitrarily). Coercion, however, must be handled as in section 2.

Example

Let Σ consist of the first groups of profiles listed for the division and multiplication operators as examples of section 4. Then $\hat{\Sigma}$ consists of all four groups listed.

6 Function Definition

Generator inductive definitions by means of the case-construct can be generalized to order sorted generator induction as follows:

For each function symbol f, with profiles in Σ_{def}, we may give one constructive definition, say $f(\bar{w}) \stackrel{\text{def}}{=} e$, where $\bar{w}$ is a list of distinct variables, and e is an expression possibly containing case constructs. For each f-profile $f : D \rightarrow T$ in Σ_{def} e must be well-formed and the minimal type of e must be included in T, assuming the type of the variable list $\bar{w}$ is D. Notice that well-formedness must be required without the use of coercion since f is to be interpreted as a total function. It is sufficient to type check a definition of f for every (non-redundant) f-profile in Σ. It then follows that the definition satisfies all f-profiles in $\hat{\Sigma}$ (and $\bar{\Sigma}$).

New variables introduced in a case construct are implicitly typed so that the discriminators are the basic restrictions of the generators, as specified by the (unique) basic profile for each generator. For the purpose of type checking, the case-construct may be considered as part of the expression language, as explained in chapter 4, its minimal type being the least common ancestor of its branches. The minimal type of an IF-expression is defined in a similar way.

The definition of a representation function may be as in section 3, but it should satisfy $rep : B \rightarrow B$ for every basic subtype B of the family under definition. This additional syntactic requirement to the definition of rep provides a guarantee that the basic types have disjoint value sets.

Examples

The generator basis of the type *Set* of the example in section 3 may be redefined as follows:

$$\Sigma_{bas}^{Set} \stackrel{\text{def}}{=} \{\emptyset :\rightarrow Eset,\ add : Set \times T \rightarrow Neset\}$$

where *Eset* and *Neset* are the basic subtypes of *Set*. It is easy to see that the representation functions defined in the example satisfy the required signature $\{rep : Eset \rightarrow Eset,\ rep : Neset \rightarrow Neset\}$.

The functions **S** and **N** introduced in examples of section 4, basic on the domains *Nat* and *Nat1*, respectively, may be extended to functions on the whole of type *Int* by defining them on the domains *Neg1* and *Neg*, respectively.

$$\begin{array}{ll} \mathbf{SNS}x = \mathbf{N}x & (\text{for } x : Nat) \\ \mathbf{N}x = \mathbf{case}\ x\ \mathbf{of}\ 0 \rightarrow 0\ []\ \mathbf{N}y \rightarrow y\ \mathbf{fo} & (\text{for } x : Neg) \end{array}$$

Notice that only the relevant branches occur for the case constructs of **S** and **N**. For the former there are two nested case constructs, both with a single non-redundant branch. Therefore, in order to simplify the syntax, the discriminations have been replaced by a Guttag-style left hand side, where the outermost function application is by definition non-basic and the inner ones are basic. Notice that the minimal type of $\mathbf{NS}x$, for x of type *Nat*, is exactly *Neg1*, the domain of the non-basic part of **S**.

The predecessor function **P** may now be defined directly in terms of the extended **S** and **N** functions.

$$\mathbf{P}x = \mathbf{NSN}x \qquad (\text{for } x : Int)$$

The division operator of section 4 may be defined convergently using an auxiliary function:

$$\begin{aligned}
&\hat{}\,\mathbf{mod}\,\hat{} : Nat \times Nat1 \rightarrow Nat \\
&x\ \mathbf{mod}\ y = \mathbf{case}\ x\ \mathbf{of}\ 0 \rightarrow 0\ []\ \mathbf{S}x \rightarrow \\
&\qquad\qquad \mathbf{if}\ \mathbf{S}(x\ \mathbf{mod}\ y) = y\ \mathbf{th}\ 0\ \mathbf{el}\ \mathbf{S}(x\ \mathbf{mod}\ y)\ \mathbf{fi}\ \mathbf{fo} \\
&x/y = \mathbf{case}\ y\ \mathbf{of}\ \mathbf{N}z \rightarrow (\mathbf{N}x)/z\ []\ \mathbf{S}z \rightarrow \\
&\qquad\qquad \mathbf{case}\ x\ \mathbf{of}\ 0 \rightarrow 0\ []\ \mathbf{N}x \rightarrow \mathbf{NS}(x/y)\ []\ \mathbf{S}x \rightarrow \\
&\qquad\qquad\qquad \mathbf{if}\ x\ \mathbf{mod}\ y = z\ \mathbf{th}\ \mathbf{S}(x/y)\ \mathbf{el}\ x/y\ \mathbf{fi}\ \mathbf{fo}\ \mathbf{fo}
\end{aligned}$$

With this definition the rest is always non-negative. Notice that the last occurrence of the expression $x\ \mathbf{mod}\ y$ is well-formed. This is because the type of y, of type *Nzero* in the left hand side, may be strengthened to *Nat1* in the latter branch of the outer case-construct. Termination of the $\hat{}/\hat{}$ definition follows by giving the second argument highest priority in the lexicographical ordering.

7 Conclusion

We have shown that the concept of order sorted algebras and the idea of generator induction can be combined in a way which brings out the benefits of both: the flexibility of typing and treatment of partial functions of the former, as well as the syntactic control with logical consistency and ground completeness of the latter. The proof technique provided by generator induction is easily generalized to the order sorted case.

In our treatment the union and intersection of related types and type products are syntactically computable. This makes it possible to detect inconsistent signatures and to automatically construct a monotonic, weakly regular and complete signature based on an arbitrary (consistent) one. This implies increased specification flexibility in the sense that signatures become extendable and composable. In contrast to [2] our signature completion is interpretation equivalent with the given one. The typing control is stronger than in OBJ, and [1], in the sense that any expression is well-formed if all its ground instances are well-formed.

Through an example we show that the trick of defining functions as basic generators on part of their domains can be useful for constructing one-to-one generator bases. Through another example we indicate a way of achieving syntactic guarantee for consistency combined with ground completeness (equalities included) for types with many-to-one generator bases.

References

[1] L. Cardelli: "A Semantics of Multiple Inheritance" In *Semantics of Data Types*, G. Kahn, Ed., Springer-Verlag, 1984. Lecture Notes in Computer Science, Volume 173.

[2] R.J. Cunningham, A.J.J. Dick: "Rewrite Systems on a Lattice of Types" Acta INFORMATICS 22, 149-169, 1985.

[3] O.-J. Dahl, D.F. Langmyhr, O. Owe: "Preliminary Report on the Specification and Programming Language ABEL." Research Report 106, Institute of Informatics, University of Oslo, Norway, 1986.

[4] O.-J. Dahl: "Object Oriented Specification." In *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner, Ed., MIT Press, 1987.

[5] K. Futasugi, J.A. Goguen, J.-P. Jouannaud, J. Meseguer: "Principles of OBJ2." In *Proceedings, 1985 Symposium on Principles of Programming Languages and Programming*, Association for Computing Machinery, 1985, pp. 52-66. W. Brauer, Ed., Springer-Verlag, 1985. Lecture Notes in Computer Science, Volume 194.

[6] J.A. Goguen: "Exception and Errors Sorts, Coercion and Overloading Operators." SRI International, Computer Science Lab, 1978.

[7] J.A. Goguen, J.-P. Jouannaud, J. Meseguer: "Operational Semantics of Order-Sorted Algebra." In *Proceedings, 1985 International Conference on Automata, Languages and Programming*, W. Brauer, Ed., Springer-Verlag, 1985. Lecture Notes in Computer Science, Volume 194.

[8] J.A. Goguen, J. Meseguer: "Order-Sorted Algebra I: Partial and Overloaded Operations, Errors and Inheritance." SRI International, Computer Science Lab, 1987.

[9] J.A. Goguen, J. Meseguer: "Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Polymorphism, and Partial Operations." Draft of 17 May 1988, SRI International, 1988.

[10] J.V. Guttag: "The Specification and Application to Programming of Abstract Data Types." Ph. D. Thesis, Computer Science Department, University of Toronto, 1975.

[11] J.V. Guttag, J.J. Horning, J.M. Wing: "Larch in Five Easy Pieces." Digital Systems Research Center, Palo Alto, California, July 1985.

[12] R. Milner: "A Proposal for Standard ML." Report CSR-157-83, Computer Science Dept., University of Edinburgh (1983). Also Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, Austin, Texas, (Aug. 1984) 184–197.