UNIVERSITY OF OSLO
Department of informatics

Parametric Subtypes in ABEL (Revised Version)

Tore Jahn Bastiansen

Research Report 207

ISBN 82-7368-118-1
ISSN 0806-3036

October 1995

UNIVERSITAS OSLOENSIS • MDCCCXI •

Parametric Subtypes in ABEL

Tore Jahn Bastiansen
Department of Informatics
University of Oslo, Norway

October 1995

Abstract

Several problems arise when parametric subtypes are used in ABEL. This paper deals with subtype parameters, the disjointness relation and the generation of profile sets, extended to handle type parameters properly. I show how more type-information can be obtained syntactically by studying the profiles of the parametric type generators.

1 Introduction

For an introduction to ABEL (Abstraction Building Experimental Language), refer to [DO91] and a more recent paper [DO95].

1.1 Types and Subtypes

Each type T in ABEL has an associated attribute V_T, where V_T is the value set of T.

There are two kinds of subtypes in ABEL; syntactic and semantic ones. Syntactic subtypes and the main type itself are defined simultaneously.

Example 1
```
type Int by Neg,Zero,Pos
         with NPos = Neg+Zero
          and Nat  = Pos+Zero
          and Nzro = Neg+Pos
==
module
  func 0 : ⇔→ Zero                                        - - zero
  func S : Nat ⇔→ Pos                                     - - successor
  func N : Pos ⇔→ Neg                                     - - negation
  one-one genbas 0,S,N                                    - - generator basis
endmodule
```

The main type is *Int*. *Neg*, *Zero* and *Pos* are called basic subtypes because they have no proper syntactic subtypes. *NPos*, *Nat*, *Nzro* are intermediate subtypes. The generator basis, G_{Int}, is specified to have a **one-one** property, which means that the type correct generator terms of type *Int* are in a one-to-one relationship with the intended abstract values (integers). *Int* is therefore called a freely generated type.

Semantic subtypes are defined by restricting an already defined type by a predicate.

Example 2
The type *Nat10* is a subtype of *Nat*:

> **type** *Nat10* == x:*Nat* **where** $x<10$
> **module** … **endmodule**

The subtype relation is defined syntactically in ABEL. That T is a subtype of U is written $T \preceq U$. The following property is ensured:

$$T \preceq U \Rightarrow V_T \subseteq V_U \tag{1}$$

This paper will only deal with syntactic subtypes. The inverse implication of (1) will then also hold.

1.2 Parametric types

A parametric type is defined by a parametric type module, on the form

> **type** $U\{T_1, T_2, ..., T_n\}$ == **module** ... **endmodule**

where $T_i, i = 1..n$ are formal type parameters.

Example 3
The type of finite sequences:

```
type Seq{T} by NESeq,ESeq ==
module
  func ε : ⇔→ ESeq                              -- empty sequence
  func ˆ⊢ˆ : Seq × T ⇔→ NESeq                    -- right append
  one-one genbas ε,ˆ⊢ˆ
endmodule
```

Note: types under definition are referred to by the type name only, parameters are implicit.

For the subtype relation the following rule of monotonicity is compatible with (1).

$$\text{MONOTY:} \quad \frac{U_i \preceq V_j, i = 1..n}{T\{U_1, U_2, ..., U_n\} \preceq T\{V_1, V_2, ..., V_n\}}$$

Let the following be a notational convention for the rest of this paper: $T\{U_1, U_2, \ldots U_n\}$ is the main type of a subtype family with subtypes T_i, $i = 1..m$. The formal type parameter list of T_i is a sublist of those of T. Since there are no generator terms of type U when

U is a formal type, we introduce a special U-token $\mathcal{T}_U$, to stand for an arbitrary value of type U. A term in T-generators and U_i-tokens is called a T-skeleton. The value set of an instantiated T consists of instantiated T-skeletons, in which U_i-tokens are replaced by V_i-values, where V_i is actual type for U_i.

2 Syntactic subtype parameters

Different types in a syntactic subtype family may have different numbers of type parameters. It is necessary to have U as a type parameter to T if U occurs in the domain of a T-generator. Otherwise, we would not be able to type formal patterns in case discriminators.

U is, however, not a necessary parameter to T if U is not in the domain of any T-generator. For the type family of *Seq*$\{T\}$, *ESeq* does not need a T-parameter, since the empty sequence, ε, is the same for all instantiations of *Seq* (See [Gus91]). For non-parametric subtypes in the same syntactic subtype family, X and Y, we have the following property:

$$X = Y \Leftrightarrow V_X = V_Y$$

Keeping parameter lists minimal ensures this property also for parametric types.

It is possible for the system to assign minimal parameter lists to uninstantiated parametric subtypes automatically. To each subtype there is assigned the list of formal types that occur in the codomain of some of its generators. It is also checked that the list of the main type is correct. The user defined parameter list for the main type is necessary, because the order and names of the formal types must be known. The order in the parameter lists of the subtypes is the same as for the main type.

If the type is recursively defined, a fixpoint algorithm is needed. Let T_i, $i = 1..n$ be the names of the types in the syntactic subtype family. Let T_i have parameter list P_i. P_i is a subsequence of the parameter list of the main type, say $\{U_1, U_2, \ldots, U_m\}$.

Initialize Set P_i to the list of all formal type parameters occur directly in the domain of a T_i-generator, for $i = 1..n$.

Iterate For $i = 1..n$ and $j = 1..m$, U_j is added to P_i if there is a generator g in G_{T_i} such that $U_j \in P_k$ for some k such that T_k occurs in the domain of g. U_j is then an implicit argument of T_k, and therefore also of T_i. This step is repeated until no change occurs.

Theorem 1 *U_j is a type parameter of T_i if and only if there is a T_i-skeleton containing a U_j-token.*

Proof: Trivial.

$\square$

The least fixpoint will often be found by the initialization, but the following example illustrates why a fixpoint algorithm is necessary.

Example 4

```
type T{U,V,W} by T1,T2,T3
       with T12 = T1+T2 and T13 = T1+T3 ==
module
  func g1 : U ⇔→ T1
  func g2 : T13 × V ⇔→ T2
  func g3 : T12 × W ⇔→ T3
  genbas g1,g2,g3
endmodule
```

The least fixpoint will be found in two passes.

	Initial lists	Pass 1	Pass 2
$T1$	$\{U\}$	$\{U\}$	$\{U\}$
$T2$	$\{V\}$	$\{U,V,W\}$	$\{U,V,W\}$
$T3$	$\{W\}$	$\{U,V,W\}$	$\{U,V,W\}$
$T12$	$\{U,V\}$	$\{U,V,W\}$	$\{U,V,W\}$
$T13$	$\{U,W\}$	$\{U,V,W\}$	$\{U,V,W\}$
T	$\{U,V,W\}$	$\{U,V,W\}$	$\{U,V,W\}$

3 The disjointness relation

ABEL has a syntactically defined disjointness relation, $\prec\!\!\succ$. Disjoint types have no common values:

$$T \prec\!\!\succ U \Rightarrow V_T \cap V_U = \emptyset \qquad (2)$$

Note that the inverse implications (2) do hold for syntactic subtypes.

The disjointness relation is useful to the type checking algorithm of ABEL. Programming errors can for instance be discovered when coercion between disjoint types would be needed.

The basic syntactic subtypes are disjoint by definition. For syntactic subtypes, disjointness is then easily checked by looking at the sets of basic subtypes included in each type. If the sets are disjoint, then so are the subtypes.

For parametric subtypes, we want to investigate whether a monotonicity rule for disjointness, like for the subtype relation, would hold. It turns out, however, that $U\{T_1\}$ and $U\{T_2\}$ may have common values even if $V_{T_1} \cap V_{T_2} = \emptyset$. The following example illustrates this:

Example 5
Consider the type of finite sequences, *Seq*$\{T\}$ of example 3. Even though *Nat* and *Neg* of example 1 are disjoint, *Seq*$\{$*Nat*$\}$ and *Seq*$\{$*Neg*$\}$ are not, because they have the value ε in common. On the other hand, *NESeq*$\{$*Nat*$\} \prec\!\!\succ$ *Seq*$\{$*Neg*$\}$.

To handle disjointness of parametric types, we introduce the concept of *disjointness preserving* formal type parameters.

Definition 1 (Disjointness preserving formal types) *U_i is a disjointness preserving formal parameter of $T\{\overline{U}\}$ if $T\{\overline{V}\}$ and $T\{\overline{W}\}$ have no common values when V_i and W_i are disjoint. $\overline{X}$ is here short for the list $X_1, X_2, \ldots, X_n$, for $X \in \{U, V, W\}$.*

Without violating (2) we can now define

$$V_i \prec\!\succ W_i \Rightarrow T\{\overline{V}\} \prec\!\succ T\{\overline{W}\}$$

when V_i and W_i are actuals for the same disjointness preserving formal parameter, since $T\{\overline{V}\}$ and $T\{\overline{W}\}$ have no common values.

We restrict ourselves in the following, to consider only types with a one-to-one generator basis. We can then compute the disjointness preserving parameters syntactically, by studying generator profiles.

Lemma 1 *The formal type U is a disjointness preserving parameter of T if and only if every T-skeleton contains at least one $\mathcal{T}_U$.*

Proof: Let V and W be actual types for U in $x : T\{\ldots, V, \ldots\}$ and $y : T\{\ldots, W, \ldots\}$. Assume that every T-skeleton contain at least one $\mathcal{T}_U$. For x and y to be equal, given that T has a one-to-one generator basis, they must be instances of the same skeleton. The latter contains at least one $\mathcal{T}_U$, which is instantiated to a V-value in x and to a W-value in y. If $V \prec\!\succ W$ then x and y cannot be equal, consequently

$$V_{T\{\ldots,V,\ldots\}} \cap V_{T\{\ldots,W,\ldots\}} = \emptyset$$

The proof the other way is trivial.

□

Lemma 2 *Every T-skeleton will contain at least one $\mathcal{T}_U$ if and only if for every type D (possibly a Cartesian product) that is the domain of a T-generator, every D-skeleton contains at least one $\mathcal{T}_U$.*

Proof: Trivial. (Note that all formal parameters of the Cartesian product type are disjointness preserving.)

□

Let $\mathcal{D}(T_i)$ be the set of formal parameters U, such that every T_i-skeleton will contain a $\mathcal{T}_U$. $\mathcal{D}$ can be computed simultaneously for all T_i, $i = 1..n$, by the following fixpoint algorithm.

Initialize Let for $i = 1..n$, $\mathcal{D}(T_i) :=$ the set of all formal types of T_i.

Iterate For $i = 1..n$, U is removed from $\mathcal{D}(T_i)$ if there is a T_i-generator with domain D such that:

1. D has no occurrences of U, or
2. for every occurrence of $V\{\ldots, U, \ldots\}$ in D, where U is actual for the formal type W, $W \notin \mathcal{D}(V)$. Note that if T is recursively defined, the old $\mathcal{D}$ is used in this analysis.

This step is repeated until no change occurs.

Theorem 2 *$\mathcal{D}(T_i)$ is the largest possible set of disjointness preserving formal type parameters of T_i.*

Proof: For each type, theorem 1 shows that any sets larger than the parameter set would be too large. Therefore, we start with sets that are large enough, and then narrow the sets until no change occurs. Thus, the algorithm finds the largest fixpoint. Note that the sets are partially well-ordered by $\subseteq$, and that the iteration step is monotonic in the sense that the sets can only decrease.

That the iteration step is correct, i.e. that the new $\mathcal{D}$ is correct according to the old $\mathcal{D}$, follows from lemma 2. From lemma 1 follows that $\mathcal{D}(T_i)$ is the largest possible set such that only disjointness preserving formal types are in $\mathcal{D}(T_i)$.

$\square$

$\mathcal{D}$ is only sufficient if we want to compute disjointness of different instantiations of the same type. For instantiations of T_i and T_j, disjointness can be computed by looking at the intersection of T_i and T_2. When T_i and T_j are in the same syntactic subtype family, the intersection of uninstantiated types are always computable, and denoted $T_i \sqcap T_j$. The following property is ensured:

$$V_{T_i \sqcap T_j} = V_{T_i} \cap V_{T_j} \tag{3}$$

Let $\mathcal{DD}(T_i, T_j)$ be a set of pairs of formal type parameter positions of T_i and T_j, defined as follows: (p, q) is in $\mathcal{DD}(T_i, T_j)$ if there is a formal type $U \in \mathcal{D}(T_i \sqcap T_j)$ and U is the formal type parameter in position p in T_i and q in T_j.

For the subtype family of *Seq*, $\mathcal{D}(\mathit{ESeq}) = \mathcal{D}(\mathit{Seq}) = \emptyset$ and $\mathcal{D}(\mathit{NESeq}) = \{T\}$. The following table illustrates $\mathcal{DD}$:

	ESeq	*NESeq*	*Seq*
ESeq	$\emptyset$	$\emptyset$	$\emptyset$
NESeq	$\emptyset$	$\{(1,1)\}$	$\{(1,1)\}$
Seq	$\emptyset$	$\{(1,1)\}$	$\emptyset$

The following inference rule can now be used to decide if to parametric types are disjoint. T_i and T_2 must be in the same syntactic subtype family:

$$\text{PARAM}_{\prec\!\succ}\!: \quad \frac{\exists\,(p,q) \in \mathcal{DD}(T_i, T_j) \bullet U_p \prec\!\!\succ V_q}{T_i\{U_1, U_2, ..., U_n\} \prec\!\!\succ T_j\{V_1, V_2, ..., V_m\}}$$

Theorem 3 *The rule PARAM$_{\prec\succ}$ is compatible with (2).*

Proof: Let $x : T_i\{U_1, U_2, ..., U_n\}$ and $y : T_j\{V_1, V_2, ..., V_m\}$. Assume that the premise of PARAM$_{\prec\succ}$ holds. We need to prove that x and y cannot be equal. For x and y to be equal, they must have equal skeletons. Let $T_\sqcap = T_i \sqcap T_j$. By (3), the skeleton must be in $V_{T_\sqcap}$, but from the definition of $\mathcal{DD}$, U_p and V_q are actuals for the same type W, and there is a $\mathcal{T}_W$

in every $T_{\sqcap}$-skeleton. Therefore x and y cannot be equal when $U_p \prec\!\succ V_q$.

□

If we want to handle types with a many-to-one generator basis, we have to consider the possibility of *redundant* formal type parameters. A formal type parameter U, is redundant if U-tokens may not be significant to the equality relation, even if there is at least one $\mathcal{T}_U$ in every T-skeleton.

It is possible to constrain the language to make redundant parameters illegal. For all type parameters U of any parametric type T, a function that extracts all U-tokens from a T-skeleton, considering all $\mathcal{T}_U$-occurrences to be mutually distinct, can be defined automatically:

$$\textbf{func}\ setU\ :\ T\{\ldots,U,\ldots\} \Leftrightarrow\!\rightarrow Set\{U\}$$

If the function $setU$ is consistent with the congruence property of the equality relation over T-values, the formal type U is not redundant, i.e.

$$x =_T y \Rightarrow setU(x) =_{Set\{U\}} setU(y)$$

If we prohibit redundant formal types, $\mathcal{D}$ and $\mathcal{DD}$ can be computed the same way for any parametric type, whether freely generated or not.

4 Parametric subtypes in function domains

Let $f : D \Leftrightarrow\!\rightarrow C$ be a function where the domain D may be a Cartesian product, and C is the codomain. A fixpoint algorithm for generating a profile set for f, is presented (see [Dah92], [OD91] and [Gus91]). The profile set $\mathcal{P}$ consists of profiles (D_i, C_i), where $D_i, i = 1..n$ are all possible (syntactic) subtypes of D (pointwise if D is a Cartesian product).

Let $T \cup U$ denote a type with value set equal to the union of the value sets of T and U. That is, $V_{T\cup U} = V_T \cup V_U$. If we restrict ourselves to non-parametric syntactic subtypes, such union operations on types can be computed syntactically by representing any type by its set of included basic subtypes. A basic subtype of a Cartesian product is a product of basic subtypes. Union operations on Cartesian products can be computed syntactically if we represent products as the sets of included basic products.

For types D_i and C_i, $i = 1..2$, the typing algorithm implies that if both $D_1 \Leftrightarrow\!\rightarrow C_1$ and $D_2 \Leftrightarrow\!\rightarrow C_2$ are valid profiles for a function f, then $D_1 \cup D_2 \Leftrightarrow\!\rightarrow C_1 \cup C_2$ is also valid. The fixpoint algorithm may then be speeded up by only considering profiles with basic domains (see [OD91]). Intermediate profiles can be generated by union operations.

Representing Cartesian products as sets of included basic products, can lead to types that are not expressible in ABEL syntax, such as the type $\{(Pos \times Neg), (Neg \times Pos)\}$. With a more straightforward representation of Cartesian products, union operations cannot be computed, but it is still possible to construct all expressible intermediate profiles.

Two types are mutually related if they have a common supertype. Let $T \sqcup U$ denote the smallest common supertype of types T and U. $T \sqcup U$ is always defined and unique if T

and U are related. Since both T and U are subtypes of $T \sqcup U$, both V_T and V_U are included in $V_{T \sqcup U}$:

$$V_{T \sqcup U} \supseteq V_T \cup V_U$$

Let $\mathcal{B}(T)$ be the set of basic subtypes included in a non-parametric type T, and $\mathcal{P}_f$ be the set of basic profiles for a function f. The codomain C, of an intermediate profile with domain D, is now valid if

$$C = \bigsqcup \{C' | D' \Leftrightarrow\!\!\!\!\rightarrow C' \in \mathcal{P}_f | D' \in \mathcal{B}(D)\} \tag{4}$$

To accept this, note the following property on $\mathcal{B}$:

$$\bigcup_{D' \in \mathcal{B}(D)} V_{D'} = V_D \tag{5}$$

Since the basic domains together span the value set of the intermediate domain D, the value of an application of $f(e)$, where $e : D$, will always be included in one of the codomains of the basic profiles.

Consider a general type expression $T\{V_1, \ldots, V_n\}$. Its basic subtypes are on the form $T'\{V_1', \ldots, V_m'\}$, where T' is a basic subtype of T and each V_j' is a basic subtype of the corresponding V_i. For instance, for sequences of non-zero integers:

$$\mathcal{B}(\mathit{Seq}\{\mathit{Nzro}\} = \{\mathit{ESeq}, \mathit{NESeq}\{\mathit{Neg}\}, \mathit{NESeq}\{\mathit{Pos}\}\}$$

The following example illustrates why (4) will give illegal intermediate profiles in this case:

Example 6
The function *sum* that computes the sum of a sequence of integers, is defined as follows:

```
func sum : Seq{Int} ⇔→ Int
def sum(q) == case q of ε → 0
                      | q'⊢x → sum(q')+x fo
```

The basic profile set will be:

```
sum:
  ESeq ⇔→ Zero
  Seq{Zero} ⇔→ Zero
  NESeq{Pos} ⇔→ Pos
  NESeq{Neg} ⇔→ Neg
```

Using equation (4) to generate an intermediate profile with domain $\mathit{NESeq}\{\mathit{Nzro}\}$ will give the codomain $\mathit{Pos} \sqcup \mathit{Neg} = \mathit{Nzro}$, but the profile

$$\mathit{NESeq}\{\mathit{Nzro}\} \Leftrightarrow\!\!\!\!\rightarrow \mathit{Nzro}$$

is not valid.

The problem with the sequence type is that (5) does not hold. The union $\mathit{NESeq}\{\mathit{Neg}\} \cup \mathit{NESeq}\{\mathit{Pos}\}$ is the type of (nonempty) sequences where either all elements are negative numbers or all are positive. This is a smaller type than $\mathit{NESeq}\{\mathit{Neg}\} \sqcup \mathit{NESeq}\{\mathit{Pos}\} = \mathit{NESeq}\{\mathit{Nzro}\}$.

Definition 2 (Divisible type parameter) *Let $T\{\ldots, U, \ldots\}$ be an uninstantiated type. U is said to be divisible for T if and only if: For an arbitrary actual type W for U,*

$$\bigcup_{W' \in \mathcal{B}(W)} V_{W'} = V_W \Rightarrow \bigcup_{W' \in \mathcal{B}(W)} V_{T\{\ldots,W',\ldots\}} = V_{T\{\ldots,W,\ldots\}}$$

A Cartesian product of length n, can be viewed as a parametric type with n formal type parameters, all divisible (can be proven). The following theorem states which type parameters are divisible for an arbitrary parametric type.

Theorem 4 *A formal type parameter U of $T\{\ldots, U, \ldots\}$ is divisible if and only if no T-skeleton can contain more than one U-token.*

Proof: Let N_U be the largest possible number of U-tokens in a T-skeleton (possibly infinite). Let W be actual type for U in $T\{\ldots, W, \ldots\}$. Every $T\{\ldots, W, \ldots\}$-value is such that for every $w : W$ it contains, there is exactly one type $W' \in \mathcal{B}(W)$ such that $w : W'$. If $N_U = 1$ then every $t : T\{\ldots, W, \ldots\}$ is also of type $T\{\ldots, W', \ldots\}$ for the same W' as above. U is therefore divisible:

$$t \in V_{T\{\ldots,W,\ldots\}} \Rightarrow t \in \bigcup_{U_i' \in \mathcal{B}(U_i)} V_{T\{\ldots,W',\ldots\}}$$

If $N_U > 1$ then t may contain different W-values included in different types in $\mathcal{B}(W)$, t is then not in $V_{T\{\ldots,W',\ldots\}}$ for any type $W' \in \mathcal{B}(W)$

$\square$

It is possible to syntactically distinguish divisible parameters form non-divisible ones, only by looking at the generator basis of a parametric type.

Theorem 5 *For a parametric type $T\{\ldots, U, \ldots\}$, U is divisible if and only if the following holds for the domain of every T-generator:*

1. *there is at most one occurrence of U, and*
2. *U does not occur, directly or indirectly, as a parameter to any V in a non-divisible position.*

Proof: This simple proof can be done by induction on the syntactic complexity of generator terms.

$\square$

Note that to check the second property, a fixpoint algorithm is needed when T and V are mutually related.

Nzro in $\mathit{NESeq}\{\mathit{Nzro}\}$ cannot be divided into basic subtypes, because the type $\mathit{NESeq}\{T\}$ has a generator with two occurrences of T in its domain.

func $\hat{}\vdash\hat{}$: $\mathit{Seq}\{T\} \times T \Leftrightarrow\!\!\rightarrow \mathit{NESeq}\{T\}$

On the other hand, the Cartesian product type has only one occurrence of each formal type in the domain of its single generator.

func $(\hat{}, \ldots, \hat{})$: $T_1 \times \ldots \times T_2 \Leftrightarrow\!\!\rightarrow (T_1 \times \cdots \times T_n)$

To make a profile set for the function *sum*, one possible solution is to leave the *Int*-parameter undivided at the expense of type information. We will then get the profile set:

sum:
$\mathit{ESeq} \Leftrightarrow\!\!\rightarrow \mathit{Zero}$
$\mathit{NESeq}\{\mathit{Int}\} \Leftrightarrow\!\!\rightarrow \mathit{Int}$

It is, however, possible to get more type-information by using *semi-basic* profile sets:

Definition 3 (Semi-basic types) *A type $T\{U_1, U_2, \ldots, U_n\}$ is called semi-basic if and only if T is basic, and, for $i = 1..n$, U_i is semi-basic when U_i is divisible.*

The function $\mathcal{B}(T)$ is redefined to give the set of semi-basic subtypes included in T. Intermediate profiles are generated from semi-basic ones by (4), with the new definition of $\mathcal{B}$.

Example 7
The function *sum* (from example 6) will get the following semi-basic profile set.

sum:
$\mathit{ESeq} \Leftrightarrow\!\!\rightarrow \mathit{Zero}$
$\mathit{NESeq}\{\mathit{Zero}\} \Leftrightarrow\!\!\rightarrow \mathit{Zero}$
$\mathit{NESeq}\{\mathit{Pos}\} \Leftrightarrow\!\!\rightarrow \mathit{Pos}$
$\mathit{NESeq}\{\mathit{Neg}\} \Leftrightarrow\!\!\rightarrow \mathit{Neg}$
$\mathit{NESeq}\{\mathit{Nat}\} \Leftrightarrow\!\!\rightarrow \mathit{Nat}$
$\mathit{NESeq}\{\mathit{NPos}\} \Leftrightarrow\!\!\rightarrow \mathit{NPos}$
$\mathit{NESeq}\{\mathit{Nzro}\} \Leftrightarrow\!\!\rightarrow \mathit{Int}$
$\mathit{NESeq}\{\mathit{Int}\} \Leftrightarrow\!\!\rightarrow \mathit{Int}$

We have $\mathcal{B}(\mathit{Seq}\{\mathit{Nzro}\}) = \{\mathit{ESeq}, \mathit{NESeq}\{\mathit{Pos}\}, \mathit{NESeq}\{\mathit{Neg}\}, \mathit{NESeq}\{\mathit{Nzro}\}\}$. The codomain of an intermediate profile with domain $\mathit{Seq}\{\mathit{Nzro}\}$ can now be obtained from the codomains the four corresponding profiles.

$$\mathit{Zero} \sqcup \mathit{Pos} \sqcup \mathit{Neg} \sqcup \mathit{Int} = \mathit{Int}$$

and we get the valid profile:

$$\mathit{Seq}\{\mathit{Nzro}\} \Leftrightarrow\!\!\rightarrow \mathit{Int}$$

The profile sets generated are monotonic (in the covariant sense). That is, if we have two profiles for a function f, $D_1 \Leftrightarrow\!\!\rightarrow C_1$ and $D_2 \Leftrightarrow\!\!\rightarrow C_2$, then $D_1 \preceq D_2 \Rightarrow C_1 \preceq C_2$. Therefore, if D_1 and D_2 is in $\mathcal{B}(D_3)$ and $D_1 \preceq D_2$, D_1 is redundant in the sense that the C_1 will add nothing to the codomain when generating a profile with domain D_3 (by equation (4)). For efficiency reasons, we can then define $\mathcal{B}(T)$ only to contain only non-redundant semi-basic (sub)types. $\mathcal{B}(\mathit{Seq}\{\mathit{Nzro}\})$ is then equal to $\{\mathit{ESeq}, \mathit{NESeq}\{\mathit{Nzro}\}\}$

The fixpoint algorithm will still have to iterate over all semi-basic subtypes, but intermediate profile generation will be more efficient.

5 Conclusion

The subtype mechanism in ABEL becomes more complicated when parametric types are used. I have investigated problems imposed by such types, and have shown how these problem can partially be solved by syntactic analysis. Notice that the solutions to the problems discussed here are very similar. Formal types are classified according to possible number of occurrences of their values, and in all three cases, fixpoint analysis of generator profiles is necessary.

In contrast to [OD91], the profile set generation algorithm is not dependent on union operations being syntactically computable.

Acknowledgments

Ole-Johan Dahl and Olaf Owe have given valuable comments through careful reading of earlier versions.

References

[Dah92] Ole-Johan Dahl. *Verifiable Programming.* Prentice Hall International, England, 1992.

[DO91] Ole-Johan Dahl and Olaf Owe. Formal development with ABEL. In S. Prehn and W.J. Toetene, editors, *Formal Software Development Methods, LNCS 552*, pages 320–362. VDM'91, Springer, 1991.

[DO95] Ole-Johan Dahl and O. Owe. On the use of subtypes in ABEL. Research Report 206, ISBN 82-7368-117-3, Department of informatics, University of Oslo, Norway, 1995.

[Gus91] Bente Gustavsen. Forbedret typeanalyse i ABEL. Master's thesis, Department of Informatics, University of Oslo, Norway, 1991. In Norwegian.

[OD91] Olaf Owe and Ole-Johan Dahl. Generator induction in order sorted algebras. *Formal Aspects of Computing*, 3:2–20, 1991.