UNIVERSITY OF OSLO
Department of informatics

On the Use of Subtypes in ABEL (Revised Version)

Ole-Johan Dahl
and Olaf Owe

Research Report 206

ISBN 82-7368-117-3
ISSN 0806-3036

October 1995

On the Use of Subtypes in ABEL

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

Abstract
ABEL is a wide spectrum language developed at the University of Oslo. The applicative core of ABEL is a typed first order language with subtypes and partial functions. The paper presents a constructive fragment of the core, based on terminating generator induction. We show how subtypes can be used to augment expressiveness and strengthen syntactic controls.

Keywords
Algebraic specification, Specification language, Generator induction, Types, Subtypes, Partial functions, Definedness, Syntactic control, Term rewriting.

1 Introduction

ABEL (Abstraction Building, Experimental Language) is a wide spectrum language together with a formal logic to be used in program development. It has been developed at the University of Oslo over a period of more than 15 years, mainly by the authors, and in close interaction with a regular student course on program specification and verification, [DLO86, DO91, Da92]. The most important sources of ideas have been as follows: SIMULA 67 (classes and subclasses), [DMN71], the LARCH and IOTA activities (generator induction), [GHW85], [Na83], and OBJ (order sorted algebras), [FGJ85].

Several implementation attempts have been made at earlier stages of the language development. They have largely depended on student activity, and did not lead to complete useful systems. At present an implementation written in Standard ML is well under way.

The applicative kernel of ABEL is a typed first order language with subtypes and recursive functions, based on a typed logic for partial functions, [Ow93], [EO93]. The so called TGI fragment deals with constructively defined types and functions. TGI stands for Terminating Generator Induction; however, one may define partial functions using explicit error indications. TGI specifications give rise to convergent rewrite rules, which enable efficient manipulation of formulas and other expressions for purposes of simplification and proof.

It has been of major concern that the language and its associated reasoning formalisms are such that consistency requirements and other proof obligations are as simple and manageable as possible. In particular we try to avoid consistency proofs, which we believe will

be very complicated in practice, by adding syntactic restrictions. Such restrictions give strengthened type analysis, and strengthened reasoning power by mechanical tools, and in addition may provide useful guidance for programmers who are not trained algebraists.

In this paper we present the TGI fragment of ABEL showing how subtypes can be used to augment expressiveness and strengthen syntactic controls in non-trivial ways. An accompanying paper will discuss how the TGI fragment could be extended with higher order functions, also constructively defined, [KD95]. Other accompanying papers will discuss module composition, including non-constructive ones, [BDK95], as well as some problems with parameterized subtypes, [Ba95].

2 The TGI fragment of ABEL

A type T in ABEL defines a pair

$$T \stackrel{\text{def}}{=} (V_T, F_T)$$

where V_T is the set (nonempty) of values of type T, and F_T is a set of function symbols associated with the type T. For every function f of an ABEL specification the user is obliged to provide a *profile*:

$$\textbf{func } f\colon\ T_1 \times T_2 \times \ldots \times T_{n_f} \longrightarrow U$$

specifying the domain $\mathcal{D}[f] = T_1 \times T_2 \times \ldots \times T_{n_f}$ and its codomain $\mathcal{C}[f] = U$, where $n_f \geq 0$ is called the arity of f. Note that constants are functions with zero arity.

The TGI fragment is quantifier-free and is based on constructively defined types and functions, which means that it can be seen as an applicative programming language. TGI stands for "Terminating Generator Induction".

The value set of a type T is defined by specifying a set of functions with codomain T as the "generator basis" of T. The generators by definition span the intended value set V_T. We take the ground generator terms of type T to be representations of the T values. In programming language terms generators thus give rise to data structures (as in ML [Mi83]). In order that the representation set be non-empty, at least one generator must have no T-argument.

Every non-generator function f is TGI defined through an equational axiom of the form

$$\textbf{def } f(x_1, x_2, \ldots, x_{n_f}) == \text{RHS}$$

where the left hand side introduces distinct variables x_i of type T_i, $i = 1, 2, \ldots, n_f$, and the right hand side is an expression in these variables, generators, and TGI defined functions. Recursion, direct and indirect, is allowed provided termination is ensured. Definition by "generator induction" is available through the use of **case**-constructs in the RHS, analogous to those of ML:

$$\textbf{case } x \textbf{ of } |_{i=1}^{n}\ g_i(y_1, \ldots, y_{n_{g_i}}) \rightarrow E_i \textbf{ fo}$$

where the *discriminand* x is a variable of a type T with the generator basis $\{g_1, \ldots, g_n\}$, and each *discriminator*, $g_i(\ldots)$, introduces $n_{g_i} \geq 0$ new variables, typed according to the

profile of g_i and with the alternative expression E_i as their scope. **case**-constructs can be nested. An argument in the left hand side occurring as the discriminand of a **case** construct is said to be an "inductive argument".

The TGI technique of ensuring recursion termination is to restrict recursive applications to be "guarded" by induction. This means that recursion is only allowed to occur within **case** branches, and in every recursive application an inductive argument is replaced by a subterm (e.g. a variable introduced in a discriminator for that inductive argument). In the case of nested induction and/or indirect recursion this simple syntactic check may be generalized in several ways. Still, ad-hoc termination proofs may sometimes have to be provided. The termination of the function definitions occurring in the present paper is easily checked syntactically.

The form of TGI function definitions ensure logical consistency and ground completeness (with some reservations for the treatment of equality, see below). This follows from syntactic checks that the discriminators are non-overlapping and exhaustive in every **case**-construct.

A generator inductive function definition can alternatively be expressed as a set of **case**-free equational axioms, in which **case** discriminators have been thrown back into the left hand sides. It is well known that such axiom sets comprise convergent sets of rules for term rewriting, and that generator terms, i.e. "values", are the irreducible ground terms. Term rewriting will thus be a very useful reasoning aid within the TGI framework.

Note that the TGI fragment of ABEL restricts **case** discriminators to be variables, i.e. those introduced in the left hand side of a function definition and in discriminators of enclosing **case** constructs. Non-variable discriminators would give rise to a kind of conditional rewrite rules. We do, however, permit **if** constructs with arbitrary Boolean test expressions. **if** constructs are treated as functions subjected to ad-hoc analysis during term rewriting, not as **case** constructs giving rise to conditional rules.

2.1 Type modules

A type definition has the following format, somewhat simplified (where a raised question mark indicates an optional phrase):

$$\begin{aligned}
\langle\text{type definition}\rangle &::= \textbf{type}\ \langle\text{type identifier}\rangle\langle\text{formal parameter part}\rangle^{?} \\
&\quad == \langle\text{type module}\rangle \\
\langle\text{formal parameter part}\rangle &::= \{\langle\text{type identifier list}\rangle\} \\
\langle\text{type module}\rangle &::= \textbf{module}\ \langle\text{type module item list}\rangle\ \textbf{endmodule}
\end{aligned}$$

The items of a type module include profiles and definitions of functions, as well as a generator basis specification. The set F_T of a type T consists of those functions which are introduced in its type module[1]. The following items are implied, even for a formal type T (including strict and non-strict equality, see below):

[1] The grouping of function symbols by type modules and modules of other kinds plays a role for function overloading, cf. [BDK95].

func $\hat{}=\hat{}$, $\hat{}\neq\hat{}$, $\hat{}==\hat{}: T \times T \longrightarrow Bool$
func if $\hat{}$ **then** $\hat{}$ **else** $\hat{}$ **fi** : $Bool \times T \times T \rightarrow T$
def **if** b **then** x **else** y **fi** == **case** b **of** $true \rightarrow x \mid false \rightarrow y$ **fo**

The type *Bool* is predefined with the generator basis $\{false, true\}$ and the standard set of operators.

There are several kinds of modules in ABEL which are not mentioned here, including non-constructive ones. In this connection the reader is referred to [DLO86, DO91] or to [BDK95]. The latter includes some recent language improvements pertaining to the composition of compound modules.

Example 1
type *Nat* == – natural numbers
module
  func 0: $\longrightarrow Nat$, **S**$\hat{}$: $Nat \longrightarrow Nat$ - - zero and successor
  one-one genbas 0, **S**$\hat{}$ - - generator basis
  func $\hat{}+\hat{}: Nat \times Nat \longrightarrow Nat$ - - addition operator
  def $x+y$ == **case** y **of** $0 \rightarrow x \mid \mathbf{S}y' \rightarrow \mathbf{S}(x+y')$ **fo**
  - - - - - - - -
endmodule

The generator basis of *Nat* is specified to have the **one-one** property, which informally means that generator terms of type *Nat*: 0, **S**0, **SS**0, ..., are in a one-to-one relationship with the intended "abstract" values.

Formally a generator basis specification for a type T, **genbas** $g_1, \ldots, g_n$, introduces an induction proof rule in the underlying logic:

$$\frac{P^x_{x_{i,k}}, \text{ for each } k \text{ s.th. } x_{i,k}:T \ \vdash P^x_{g_i(x_{i,1},\ldots,x_{i,n_{g_i}})}; \text{ for } i = 1,2,\ldots,n}{\vdash \forall x:T \bullet P}$$

for fresh variables $x_{i,k}$

There is one premise for each T-generator, and for each premise there is one induction hypothesis for each generator argument of type T (zero or more).

A **one-one** clause asserts that equality over T is the same as syntactic equality of generator terms, up to equality of subterms of other types occurring in the generator domains. A TGI definition to that effect is synthesized mechanically:

def $x=y$ == **case** (x,y) **of** $|_{i=1}^{n}\ (g_i(\bar{x}_i), g_i(\bar{y}_i)) \rightarrow \bar{x}_i = \bar{y}_i \mid$ **others** $\rightarrow$ *false* **fo**

where the **others** branch is syntactic sugar for the remaining generator combinations. This definition satisfies all logical requirements to an equality relation, including substitution laws.

Example 2
type *Seq*{*T*} == - - finite sequences of values of an unspecified type
module
  func ε: $\longrightarrow Seq$ - - empty sequence

func $\hat{}\vdash\hat{}: Seq \times T \longrightarrow Seq$ - - append right
func $\hat{}\dashv\hat{}: T \times Seq \longrightarrow Seq$ - - append left
func $\hat{}\vdash\!\!\dashv\hat{}: Seq \times Seq \longrightarrow Seq$ - - concatenate
one-one genbas ε, $\hat{}\vdash\hat{}$ - - a traditional choice, [Da77]
def $x \dashv q ==$ **case** q **of** $\varepsilon \rightarrow \varepsilon \vdash x \mid q' \vdash y \rightarrow (x \dashv q') \vdash y$ **fo**
def $q \vdash\!\!\dashv r ==$ **case** r **of** $\varepsilon \rightarrow q \mid r' \vdash x \rightarrow (q \vdash\!\!\dashv r') \vdash x$ **fo**
endmodule

Within a type module the type under definition can only be referred to through the family name; the formal parameter list is implicit.

The assumption that an equality operator exists for the formal type T is necessary for the equality definition induced by the **one-one** specification to be meaningful:

def $q = r ==$ **case** (q, r) **of** $(\varepsilon, \varepsilon) \rightarrow true \mid (q' \vdash x, r' \vdash y) \rightarrow q' = r' \wedge x = y$
$\mid$ **others** $\rightarrow false$ **fo**

Certain types, such as those of finite sets, have no one-to-one generator basis. Then the intended abstract values correspond to equivalence classes (or rather congruence classes) of generator terms. In order to completely define such a type, an explicit TGI definition of the equality operator may be given, or a so called "observation basis" may be specified consisting of functions able to see all observable properties of the abstract values, and nothing more.

Example 3
type $Set\{T\} ==$ - - finite sets of values of an unspecified type
module
func $\emptyset: \longrightarrow Set$ - - empty set
func $add: Set \times T \longrightarrow Set$ - - add one element
genbas $\emptyset$, add - - many-to-one generator basis
func $\hat{}\in\hat{}: T \times Set \longrightarrow Bool$ - - membership relation
def $x \in s ==$ **case** s **of** $\emptyset \rightarrow false \mid add(s', y) \rightarrow x = y \vee x \in s'$ **fo**
func $\hat{}\subseteq\hat{}: Set \times Set \longrightarrow Bool$ - - inclusion relation
def $s \subseteq t ==$ **case** s **of** $\emptyset \rightarrow true \mid add(s', x) \rightarrow x \in t \wedge s' \subseteq t$ **fo**
def $s = t == s \subseteq t \wedge t \subseteq s$
lma obsbas $\hat{}\in\hat{}$ - - sets are equal iff they have the same elements
endmodule

Here an observation basis specification is rendered in the form of a *lemma*. There is then an obligation to prove that the induced alternative definition of the equality operator:

def $s = t == \forall x : T \bullet (x \in s) = (x \in t)$

would be equivalent to the one given. The alternative definition is outside the TGI framework, but may nonetheless be useful for reasoning purposes.

Any equality definition with respect to a many-to-one generator basis, say of a type T, entails an obligation to prove that the logical properties of equality over T are not violated. An observation basis necessarily defines an equivalence relation. Still, function definition by induction over T reveals the detailed structure of the set of generator terms, not only the set of equivalence classes. Thus, there is an associated obligation to prove that the substitution property holds for any argument inductive over T: $x \stackrel{T}{=} y \Rightarrow f(\ldots,x,\ldots) = f(\ldots,y,\ldots)$. (Also generators must be checked in this way.) In a later subsection we point out a way around this difficulty.

2.2 Partial functions

So far all TGI definable functions are total ones. Partial functions may, however, be defined within the TGI framework through the introduction of an explicit "bottom" symbol $\perp$ which stands for an "ill-defined" expression, i.e. an expression with no value.

Example 4
func $\hat{}\dot{-}\hat{}: Nat \times Nat \longrightarrow Nat$ - - partial subtraction operator
def $x \dot{-} y ==$ **case** y **of** $0 \rightarrow x \mid \mathbf{S}y' \rightarrow$
case x **of** $0 \rightarrow \perp \mid \mathbf{S}x' \rightarrow x' \dot{-} y'$ **fo fo**

The TGI fragment is intended to model mathematical structures containing general recursive functions, possibly partial. Expressions in such functions either evaluate to well defined values, or the evaluation does not terminate. In the TGI framework the former case corresponds to an evaluation producing a generator term, whereas non-termination is modeled by the result $\perp$ (obtained in finite time!). Consequently TGI value domains are flat, and generators model strict (and total) functions. Furthermore **case**-constructs are strict in the discriminand, which implies that only functions monotonic with respect to definedness can be user defined.

There are two equality operators in the language, $\hat{}=\hat{}$ and $\hat{}==\hat{}$, strict and non-strict respectively. The latter gives the result *true* if the operands are defined and equal, or if both are ill-defined, otherwise *false*. Given a TGI definition of the former, the non-strict "strong" equality can be implemented constructively behind the scenes.

Term rewriting naturally leads to lazy semantics providing for non-strict functions. This is useful, and sometimes necessary as in the case of certain logical operators. The strictness requirements mentioned above will not in general be respected using ordinary TGI term rewriting. In fact, if the term u is a reduction of t, then the former may be better defined: $t \sqsubseteq u$. If, however, t is well-defined $t == u$ holds. A strongly correct and convergent rule set can be obtained mechanically from a set of TGI function definitions, but at the expense of some loss of reasoning power and efficiency. We return to this topic in section 4.

3 Subtypes

The ABEL concept of subtype is introduced as a means to:

- make expression typing stronger and more flexible,
- introduce natural function overloading for taking advantage of special cases,
- aid in the use of partial functions,
- aid in defining types not freely generated,
- introduce implementation related capacity constraints.

We introduce the following relations: $=$ (equal), $\prec$ (subtype), and $\preceq$ (subtype or equal) on types, or more precisely on ABEL type expressions:

$$\begin{aligned} \langle\text{type expression}\rangle &::= \langle\text{type identifier}\rangle\langle\text{actual parameter part}\rangle^? \\ \langle\text{actual parameter part}\rangle &::= \{\langle\text{type expression list}\rangle\} \end{aligned}$$

The equality relation represents syntactic equality, and $\prec$ is the transitive closure of a syntactic relationship on types, see below. Thus, $\preceq$ is a partial order defined syntactically by a specification text. The syntax and semantics of ABEL are such that the following holds for all type expressions T and U meaningful in the context of any ABEL specification (no proof of this fact will be given here.):

$$\text{SUBTY: } T \preceq U \;\Rightarrow\; V_T \subseteq V_U \;\wedge\; F_U \subseteq F_T$$

Thus, a subtype of a type U may have a smaller value set than that of U and an extended set of function symbols. The function symbols of U are considered to be "inherited" by the subtype.

Types T_1 and T_2 are said to be *related* if they have a common supertype, that is if $T_1 \preceq T \wedge T_2 \preceq T$ for some T. The smallest common supertype of related types is unique and is denoted by $T_1 \sqcup T_2$. In general $V_{T_1} \cup V_{T_2} \subseteq V_{T_1 \sqcup T_2}$.

The subtype relation is extended pointwise to type products. This is a special case of a rule expressing a monotonicity principle for parameterized types.

$$\text{MONOTY: } T_1 \preceq T_1' \wedge \ldots \wedge T_n \preceq T_n' \Rightarrow U\{T_1, \ldots, T_n\} \preceq U\{T_1', \ldots, T_n'\}$$

The rule is consistent with SUBTY (actually $F_{U\{T_1,\ldots,T_n\}} = F_{U\{T_1',\ldots,T_n'\}}$).

The type $\oslash$ is predefined and stands for the *empty type*:

$$V_\oslash = \emptyset, \qquad F_\oslash = \langle\text{all declared functions}\rangle$$

The empty type is by definition minimal with respect to the subtype relation:

$$\oslash \preceq T \quad \text{for all declared types } T.$$

Assume $T \preceq U$. The fact that any T value belongs to V_U gives rise to the following typing rules for expressions:

- A well-typed expression of minimal type T is either
  - a variable of type T, or
  - an application of some function $f(e_1, e_2, \ldots, e_n)$, $n \geq 0$ (omitting empty parentheses and allowing mixfix operator notations), provided e_i is a well-typed expression of minimal type T_i, and a profile $f : U_1 \times U_2 \times \ldots \times U_n \longrightarrow T$ exists, such that $T_i \preceq U_i$, $i = 1, 2, \ldots, n$, and T is minimal (and unique) for such profiles.

- A **case** or **if** construct whose alternatives are well-typed expressions of related minimal types $T_1, \ldots, T_n$, such that $T = T_1 \sqcup \ldots \sqcup T_n$.
- The expression $\perp$ is well-typed and of type $\oslash$.

Remark. In the context of subtypes it may be useful to assign more than one profile to a function. For instance, if $Nat \prec Int$, as in example 5 below, then $\hat{}+\hat{}$:$Int \times Int \longrightarrow Int$ also satisfies the profile $Nat \times Nat \longrightarrow Nat$. The profile sets occurring in the context of ABEL are such that any well-typed function application has a unique minimal type, cf [OD91].

A function definition is well-typed with respect to a given profile iff its right hand side is a well-typed expression whose minimal type is included in the codomain of the function profile, given that the variables of the left hand side are typed as required by the profile. TGI function definitions are required to be well-typed with respect to all its profiles.

The fact that $\oslash$ is included in all types has the consequence that any function profile specifies a (possibly) partial function. Since $\perp$ is of type $\oslash$, it is an acceptable argument to all functions. A sufficient condition for a TGI defined function to be total is that its right hand side only contains total function applications, and does not contain $\perp$.

Expressions are *weakly typed* in the sense of the following theorem.

Theorem 1
Let e be a well-typed expression of minimal type T. Then any ground instance of e either has a value of type T (or a subtype of T) or it has no value.
Proof:
Directly by induction on the structure of e, using the typing rules, together with the fact that the function definitions are well-typed and terminating.

The theorem expresses that the minimal type of an expression is *semantically correct*. The type is said to be *optimal* if no smaller type is semantically correct for the expression. Similarly we say that a profile $f : T \longrightarrow U$ is optimal if U is the optimal type of $f(x)$ for x of type T.

Remark. The idea of weak typing corresponds to that of partial correctness of programs and is of fundamental importance in ABEL. It can lead to proof decomposition, such that aspects of well-definedness are dealt with separately. Notice that TGI term rewriting agrees well with weak typing: if a term t is of type T and u is a reduction of t, then u is of type T as well, although possibly better defined.

In practice it will often be necessary to deal with expressions which are not well-typed. This is a result of the fact that a well-typed expression of minimal type T may well have a value of a type properly included in T. Let $f : T_1 \longrightarrow U$ and $e : T_2$, and consider the expression $f(e)$. It is well-typed if $T_2 \preceq T_1$, otherwise not. Now, if T_1 and T_2 are unrelated types the expression must be considered plainly wrong, but if they are related (and not known to be value disjoint), the expression may well be semantically meaningful. We can make it well-typed by a "coercion", forcing its type to become T_1. ABEL provides two alternative coercion mechanisms:

- "Proof time coercion", e **qua** T_1, which entails an obligation to prove that in the given textual environment the value, if any, of e is necessarily of type T_1.
- "Run time coercion", e **as** T_1, which is an application of a strict and partial function to check the possible outcome of e. Let T be the maximal supertype of T_1 (and of T_2). Then the coercion function can be informally specified as follows:

  func ^**as** T_1: $T \longrightarrow T_1$
  def x **as** $T_1 ==$ **if** $x \in V_{T_1}$ **then** x **else** $\perp$ **fi** - - type correct by definition

Clearly e **qua** $T_1 == e$ and e **as** $T_1 \sqsubseteq e$.

Coercion functions from a maximal type to its subtypes are defined automatically. It is being debated whether or not coercion insertion of some kind should also be automatic. In view of the typing rules of ABEL we shall say that expressions are "coercion-free upwards" in a type hierarchy.

Let e be an arbitrary generator term of a subtype $T' \prec T$. T' is said to be *convex* if the type of any direct subterm of e related to T' is included in T'. It implies that any T' value can be built up of T-generators without stepping outside the subtype T'. Thus, if convexity has been proved for T', then any T-variable introduced in a discriminator for a type T' discriminand is known to be of type T'. This, in conjunction with function overloading in subtypes, may lead to stronger typing and less need for coercion.

ABEL supports two kinds of subtypes: "syntactic" subtypes and "semantic" ones.

3.1 Syntactic subtypes

We may define a family of syntactic subtypes by introducing a list of so called "basic subtypes", as well as "intermediate subtypes", in addition to the main type. The basic subtypes occur as codomains of generators and have disjoint value sets by definition. For a one-to-one generator basis the disjointness is a direct consequence of the one-one property; for an many-to-one basis, however, proofs of disjointness are required. All types of a syntactic subtype hierarchy have the same set of associated functions.

Example 5

```
type Int by Zero, Pos, Neg
        with Nat=Zero⊔Pos, Nzro=Neg⊔Pos, Npos=Zero⊔Neg ==
  module
    func 0: ⟶ Zero,                                          - - zero
         S^: Nat ⟶ Pos,          - - the successor of a natural is positive
         N^: Pos ⟶ Neg          - - a negated positive number is negative
    one-one genbas 0, S^, N^
  endmodule
```

Notice the domains of the unary generators. As a consequence the well-typed generator terms are in a one-to-one correspondence with the integers. For the basic subtypes we have: $V_{Zero} = \{0\}$, $V_{Pos} = \{\mathbf{S}0, \mathbf{SS}0, \ldots\}$, and $V_{Neg} = \{\mathbf{NS}0, \mathbf{NSS}0, \ldots\}$. The intermediate types *Nat*, *Nzro* and *Npos* represent the indicated type unions. *Nat*, as well as *Int* and *Zero*, are convex types. This is follows from the generator profiles.

In general a complete syntactic type hierarchy is a lattice whose maximal and minimal elements are the main type and $\oslash$, respectively. The lattice operations correspond to union and intersection of the value sets:

$$V_{T_1 \sqcup T_2} = V_{T_1} \cup V_{T_2}, \qquad V_{T_1 \sqcap T_2} = V_{T_1} \cap V_{T_2}$$

Since basic subtypes by definition have disjoint value sets, a disjointness relation, $\prec\!\!\succ$, may be syntactically defined on the elements of a syntactic subtype hierarchy, such that:

$$T_1 \prec\!\!\succ T_2 \Longleftrightarrow V_{T_1} \cap V_{T_2} = \emptyset$$

Then, obviously, $\oslash \prec\!\!\succ T$ holds for arbitrary type T.

The typing algorithm take advantage of lattice and disjointness properties in order to strengthen the type resulting from certain coercions. Let $T = T_1 \sqcup T_2$ and $U = T_2 \sqcup T_3$. If e is an expression of minimal type T, then e **qua** U can be seen to have the minimal type T_2.

3.1.1 Profile sets

Generator inductive function definitions in the context of syntactic subtypes may be subjected to a type analysis which goes deeper than just checking the validity of the user submitted function profile, say $f : T \longrightarrow U$, where T may be a Cartesian product. Thereby T is defined as the largest domain on which f can legally be applied. We consider all subtypes $T_1, T_2, \ldots, T_n$ of T and seek the subtypes $U_1, U_2, \ldots, U_n$ of U such that f applied to an argument list of type T_i has the minimal type U_i, for $i = 1, 2, \ldots, n$. Each pair (T_i, U_i) then represents an additional profile for f, $T_i \longrightarrow U_i$. We say that the result is a *minimal profile set*, $\mathcal{P}$, for f.

The typing algorithm applied to the TGI definition of f can be seen as a function $\mathcal{F}_f$ from profile sets to profile sets, as follows. Apply the algorithm to an application of f to an argument list of type T_i in context of the profile set $\mathcal{P}$. Let the type of the application be U_i'. Define the i'th component of the output profile set $\mathcal{P}'$ to be $T_i \longrightarrow U_i'$, $i = 1, 2, \ldots, n$. Then, clearly, the minimal profile set is a fixed point of $\mathcal{F}_f$. Furthermore, it is the *least* fixpoint wrt. the following order of f-profile sets:

$\mathcal{P} = \{T_i \longrightarrow U_i \mid 1 \leq i \leq n\}$ is less than or equal to $\mathcal{P}' = \{T_i \longrightarrow U_i' \mid 1 \leq i \leq n\}$
iff $\forall i : \{1..n\} \bullet U_i \preceq U_i'$.

It is easy to see that $\mathcal{F}_f$ is monotonic wrt. this ordering. Consequently, since the set of n-tuples of subsets of U is finite, the optimal profile set can be obtained as $\mathcal{P}_k = \mathcal{F}_f^k(\mathcal{P}_0)$ for some natural number k, where $\mathcal{P}_0$ is the set of profiles whose codomains are all equal to $\oslash$.

As a final step the minimal profile set may be simplified by deleting all "redundant" profiles. A profile P, $X \longrightarrow Y$, is redundant in the set $\mathcal{P}$ iff there is a different profile P', $X' \longrightarrow Y'$, in $\mathcal{P}$ such that $X \preceq X'$ and $Y' \preceq Y$. In that case P is said to be covered by P'. The resulting minimal profile set is unique, because the "covered by" relation is transitive. It also satisfies a regularity condition sufficient to secure unique typing of expressions, see [OD91].

Example 6

Let *Bool* have the basic subtypes *False* and *True*, where $V_{False} = \{false\}$ and

$V_{True} = \{true\}$, and let *Nat* be as defined in example 5. Consider the less than or equal predicate on *Nat* values:

func $\hat{} \leq \hat{}$: $Nat \times Nat \longrightarrow Bool$
def $x \leq y ==$ **case** x **of** $0 \rightarrow true \mid \mathbf{S}x' \rightarrow$
case y **of** $0 \rightarrow false \mid \mathbf{S}y' \rightarrow x' \leq y'$ **fo fo**

The fixpoint algorithm works as shown by the table below. It is sufficient to list only "basic" profiles, i.e. profiles whose domains consist of basic subtypes or $\oslash$. "Intermediate" profiles can be obtained by union operations.

	$\mathcal{P}_0$	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$
$\oslash \times \oslash \longrightarrow$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
$\oslash \times Zero \longrightarrow$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
$\oslash \times Pos \longrightarrow$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
$Zero \times \oslash \longrightarrow$	$\oslash$	$True$	$True$	$True$
$Zero \times Zero \longrightarrow$	$\oslash$	$True$	$True$	$True$
$Zero \times Pos \longrightarrow$	$\oslash$	$True$	$True$	$True$
$Pos \times \oslash \longrightarrow$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
$Pos \times Zero \longrightarrow$	$\oslash$	$False$	$False$	$False$
$Pos \times Pos \longrightarrow$	$\oslash$	$\oslash$	$Bool$	$Bool$

Sine $\mathcal{P}_2 = \mathcal{P}_3$ this is the minimal fixpoint. The inclusion of $\oslash$ among the subtypes of *Nat* gives rise to strictness analysis. Thus, $\hat{} \leq \hat{}$ is strict in its first argument, but not in its second argument as indicated by the profile $Zero \times \oslash \longrightarrow True$.

Notice the change from $\oslash$ to *Bool* in the codomain of the last profile in the list. The domain corresponds to the RHS expression $x' \leq y'$, where, according to the profile of $\mathbf{S}\hat{}$ both variables are of the type *Nat*. Consequently the codomain in $\mathcal{P}_2$ must be the type union of all the codomains in $\mathcal{P}_1$. (Due to definedness monotonicity profiles with domains containing $\oslash$ may be omitted from the union.)

By including all intermediate profiles and then omitting the redundant ones we end up with the following smallest minimal profile set for $\hat{} \leq \hat{}$:

$\oslash \times Nat \longrightarrow \oslash$
$Zero \times Nat \longrightarrow True$
$Pos \times \oslash \longrightarrow \oslash$
$Pos \times Zero \longrightarrow False$
$Nat \times Nat \longrightarrow Bool$

The profile set of a function can be interpreted as the profiles for a set of distinct overloaded functions, defined on different domains, and whose semantics coincide on common domain parts. The typing algorithm may for every f application select the function to be applied on the basis of the argument types. Since only well-typed expressions are included in the language, no function will ever be applied to arguments outside its domain.

Functions may consequently be considered *undefined* (rather than ill-defined) outside their domains, user specified as well as those of "subordinate" profiles. (This has been pointed out by Bjørn Kristoffersen.)

Under certain conditions the above algorithm produces an *optimal* profile set, i.e. a set of optimal profiles. Theorem 2 below expresses sufficient conditions for optimality, provided that **case**-free versions of function definitions are used. We assume in the following that basic subtypes are minimal, i.e. that generators have mutually distinct codomains.

- A function is said to be *base-preserving* if its set of profiles, including redundant ones, contains a basic or empty codomain for each "basic domain", i.e. one consisting of basic subtypes or $\emptyset$. **if** constructs can be seen as base-preserving functions.
- An O-term is either
  - $\perp$, or
  - a variable, or
  - a generator application (with typing capturing error propagation), or
  - a base-preserving function applied to a list of O-terms, in which no variable occurs more than once (not adding occurrences in different **if**-branches), or
  - an application of an optimally typed function (i.e. a function with an optimal profile set) to arguments consisting of unreplicated variables and generators, such that every list of arguments to a generator either matches the generator domain exactly or contains $\perp$.
- An O-function is a function whose right hand sides are O-terms when functions under definition are taken as optimally typed. (Right hand sides can be non-TGI in the sense that recursion is not guarded by induction.) Equality on a formal type can be considered an O-function, provided profiles expressing strictness properties are included.

Theorem 2. *The minimal type of an O-term is optimal, and a minimal profile set for an O-function, computed by the fixpoint algorithm, is optimal.*

Proof: By induction on syntactic structure on terms.

All TGI defined functions occurring in the examples of this paper are optimally typed (or would be if basic types were introduced for each generator). It is possible to strengthen the theorem somewhat by modifying the fixpoint algorithm to recognize optimal or base-preserving applications of general functions.

3.2 Semantic subtypes

A semantic subtype definition has the following format, somewhat simplified (using square brackets as meta-parentheses):

⟨subtype definition⟩ ::= ⟨type id⟩⟨formal parameters⟩$^?$ ==⟨module prefix⟩
⟨subtype module⟩$^?$
⟨module prefix⟩ ::= [⟨variable⟩ :]$^?$⟨type expression⟩⟨where clause⟩$^?$
⟨where clause⟩ ::= **where**⟨Boolean expression⟩[**convex**]$^?$

where the formal parameters and the subtype module, as well as the variable declaration and the where clause of the module prefix, are optional. The latter may be followed by the keyword **convex** to indicate convexity of the defined subtype. A subtype module is like a type module syntactically, but certain semantic restrictions apply.

In the subtype definition

$$\textbf{type}\ T\{T_1,\ldots,T_n\} == x{:}U\{U_1,\ldots,U_m\}\ \textbf{where}\ R(x)\ M$$

U is a defined type (family) and $U_1,\ldots,U_m$, if any, are type expressions in the formal parameters and defined types. Any instance $\bar{T}$ of the left hand side is by definition a semantic subtype of the corresponding instance $\bar{U}$, such that $V_{\bar{T}} = \{x \in V_{\bar{U}} \mid R(x)\}$, where $R \in F_{\bar{U}}$. $F_{\bar{T}}$ inherits the functions in $F_{\bar{U}}$ and contains in addition those introduced in the subtype module instance $\bar{M}$. These rules are consistent with the general SUBTY rule.

For any subtype U' of $\bar{U}$ (other than $\bar{T}$) a corresponding semantic subtype of U', denoted $U'.\bar{T}$, restricted by R, and extended by $\bar{M}$, is automatically defined. Monotonicity clearly applies to such implicit types: $X' \preceq X \wedge Y' \preceq Y \Rightarrow X'.Y' \preceq X.Y$.

Notice that $\hat{}\prec\hat{}$ is an entirely syntactic relation. Let T be a defined type. Then the definitions

$$\textbf{type}\ U == T \qquad \text{and} \qquad \textbf{type}\ V == x{:}T\ \textbf{where}\ R(x)$$

would establish U and V as new distinct types, such that the relationships $U \prec T$ and $V \prec T$ would hold, but not $V \prec U$.

A subtype module may contain a *syntactic redefinition* of any function $f : D \longrightarrow C$ associated with the supertype, by providing a function profile $f : D' \longrightarrow C'$ in which each occurrence of any of its supertypes is replaced by the subtype. It may happen that the old semantic definition, reinterpreted in the subtype module, can be shown to satisfy the new profile, mechanically by the typing algorithm or otherwise. If that is not the case, a new definition must be provided. If $D' \prec D$ a new definition may take advantage of that fact. In any case, the redefined function, say f', must be an approximation to the properly restricted old function: $f' \sqsubseteq f/D'$.

Example 7
We define a semantic subtype of *Nat* as defined in example 5, bounded by an unspecified number n.

type *BNat* == x : *Nat* **where** $x \leq n$ **convex module**
  func 0 : $\longrightarrow$ *BNat.Zero*
  def 0 == 0 **at** *Nat* **qua** *BNat*
  func **S**$\hat{}$: *BNat* $\longrightarrow$ *BNat.Pos*
  def **S**x == (**S**x) **at** *Nat* **as** *BNat*
  func $\hat{}+\hat{}$, $\hat{}-\hat{}$: *BNat* $\times$ *BNat* $\longrightarrow$ *BNat*
endmodule

The convexity of *BNat* is a syntactic consequence of the restricted generator profiles. The semantic redefinition of 0 is redundant, but has been included in order to show the obligation to prove that 0 satisfies the restriction predicate.

(The proof is trivial using the $\hat{}\leq\hat{}$ definition of example 6.) The **at**-construct binds the main operator of the preceding expression to the one associated with the indicated module. Notice the coercion test applied to the redefined successor function. The definitions of $\hat{}+\hat{}$ and $\hat{}-\hat{}$ associated with *Nat*, interpreted in the *BNat* module, satisfy the new profiles. This fact is due to the convexity of *BNat* and will be established by the typing algorithm.

Our next example shows how a semantic subtype can be used to establish a one-to-one property on top of a many-to-one generator basis, thereby removing the danger of logical inconsistency caused by generator induction over that basis. At the same time a simpler definition of equality can be given.

Example 8
Consider the concept $Set\{T\}$, as defined in example 3. A one-to-one property is established by restricting the set of generator terms to "canonical" terms of the form $add(\ldots add(add(\emptyset, a_1), a_2), \ldots, a_n)$, where $a_1 < a_2 < \ldots < a_n,\ n \geq 0$.

We construct a type of canonical sets in two steps: First an intermediate subtype $SSet\{T\}$ is defined introducing a predicate *canonic*. For that purpose it must be assumed that the formal type T has a total order $\hat{}<\hat{}: T \times T \longrightarrow Bool$. We do not here explain how such assumptions can be formalized in ABEL, but refer the reader to a companion paper.

type $SSet\{T\}\langle$assuming $\hat{}<\hat{}$ is a total order on $T\rangle$ $==$ $Set\{T\}$**module**
  func $canonic: Set \longrightarrow Bool$
  def $canonic(s) ==$ **case** s **of** $\emptyset \rightarrow true \mid add(s', x) \rightarrow$
    case s' **of** $\emptyset \rightarrow true \mid add(q'', y) \rightarrow canonic(q') \wedge y < x$ **fo fo**
endmodule

Remark: The definition of *canonic* actually introduces inconsistency, since the very purpose of the predicate is to distinguish between generator terms belonging to the same equivalence class. For instance, the terms $add(\emptyset, a)$ and $add(add(\emptyset, a), a)$ are semantically equal, both representing the singleton set $\{a\}$; but only the former is canonical. However, provided the only use of the predicate is to define a semantic subtype this inconsistency will not represent a problem.

type $CSet\{T\} == s : SSet\{T\}$ **where** $canonic(s)$ **convex module**
  func $\emptyset \longrightarrow CSet$
  func $add: CSet \times T \longrightarrow CSet$
  def $add(s, x) ==$ **case** s **of** $\emptyset \rightarrow add(s, x)$ **at** $Set \mid add(s', y) \rightarrow$
    if $y < x$ **then** $add(s, x)$ **at** Set **else if** $y = x$ **then** s
    else $add(add(s', x), y)$ **at** Set **fi fi fo qua** $CSet$
  one-one
  - - - - - - - - -
  endmodule

The discriminators of a **case** construct by definition refer to generators, not to possible redefined functions. In the last alternative of the redefinition of *add* the innermost *add* application is a recursive one, whereas the outermost one refers to the *Set* generator. Notice the **qua**-coercion. The validity of the redefinition must be proved:

$$\forall s\!:\! CSet \bullet\ add(s,x)\ \mathbf{at}\ CSet \sqsubseteq add(s,x)\ \mathbf{at}\ Set$$

(based on the equality relation of the *Set* type). The proof shows that the redefined *add* actually generates abstract *Set* values. Inspection shows that the function is total, thus the set of all such values is spanned.

The **one-one** specification asserts a one-to-one relation between *CSet* generator terms and abstract sets. Equality is redefined accordingly:

$$\begin{array}{l}\mathbf{func}\ \hat{}=\hat{}: CSet \times CSet \longrightarrow Bool\\ \mathbf{def}\quad s=t == \mathbf{case}\ (s,t)\ \mathbf{of}\ (\emptyset,\emptyset) \rightarrow true\\ \qquad\qquad\qquad\qquad\qquad\quad \mid\ (add(s'x), add(t',y)) \longrightarrow s'=t' \wedge x=y\\ \qquad\qquad\qquad\qquad\qquad\quad \mid\ \mathbf{others} \rightarrow false\ \mathbf{fo}\end{array}$$

The required verification of this redefinition consists in proving:

$$\forall s,t\!:\! CSet \bullet\ (s=t)\ \mathbf{at}\ CSet == (s=t)\ \mathbf{at}\ Set$$

The proof amounts to showing that the one-to-one property expressed by the redefinition actually holds.

The generator redefinitions ensure that copies of all function definitions given in the supertype will satisfy restricted profiles, including any set producing function. One may, however, choose to replace such definitions by more efficient ones in the algorithmic sense, taking the canonical structure of value representations into account. For instance:

$$\begin{array}{l}\mathbf{func}\ \hat{}\subseteq\hat{}: CSet \times CSet \longrightarrow Bool\\ \mathbf{def}\quad s\subseteq t == \mathbf{case}\ s\ \mathbf{of}\ \emptyset \rightarrow true \mid add(s',x) \rightarrow\\ \qquad\qquad \mathbf{case}\ t\ \mathbf{of}\ \emptyset \rightarrow false \mid add(t',y) \rightarrow x=y \wedge s'\subseteq t' \vee x<y \wedge s\subseteq t'\ \mathbf{fo}\end{array}$$

Definitions not taking canonicity into account, such as those of the *Set* module, will as a rule be easier to formulate and understand, as well as more efficient for reasoning purposes, although less efficient computationally.

4 Definedness Control

We return to the problem of implementing generator strictness and strictness in **case** discriminands by a modified set of TGI rewrite rules. For that purpose, and behind the scenes, we shall let the ill-defined expression symbol $\perp$ play the role of an additional

generator for every type. A type T so extended is denoted $T_\perp$. Then discriminand strictness is implemented by extending every **case** construct in an ABEL text by an implicit branch $\perp \rightarrow \perp$.

For a type whose generators are all constants generator strictness is not an issue, therefore regarding $\perp$ as an additional generator causes no problem. Consider, however, a type T containing a non-constant generator g. Then generator strictness requires $g(\ldots, \perp, \ldots) == \perp$ to hold. Thus, if T has a one-to-one generator basis that property is lost for $T_\perp$. Furthermore, if generator strictness axioms are included as rewrite rules in addition to a TGI rule set, then rewrite confluence is lost. To wit: a discriminand of the form $g(\ldots, \perp, \ldots)$, would match the g discriminator unless the strictness rule is applied first.

The following solution to these problems can be applied mechanically:

- Define a semantic subtype T' of $T_\perp$ whose generator terms have one of the following canonical forms: $\perp$ or $\perp$-free terms. T' is convex.
- TGI redefine each non-constant T-generator to satisfy the properly restricted profiles, and deny user access to the original generators (except in **case** discriminators).

Theorem 3. *If T has a one-to-one generator basis, then so has T' obtained from T as explained. And the set of TGI rewrite rules, extended by those redefining non-constant T-generators, is convergent.*

Proof: The one-to-one property follows from the form of the canonical value representations. Convergence is a consequence of the fact that all rewrite rules of T' are derived from TGI function definitions.

Remarks: The fact that the user is denied access to certain generators is essential for canonicity of generator terms. A construction like the one above could be applied at the user level by introducing one additional constant generator for each defined type, to represent ill-definedness for that type.

Example 9
Consider the type *Nat* as defined in example 1. Define the following predicate for $Nat_\perp$:

func *canonic*: $Nat_\perp \longrightarrow$ *Bool*
def *canonic*(x) == **case** x **of** $\perp \rightarrow$ *true* | $0 \rightarrow$ *true* | $\mathbf{S}x' \rightarrow$
case x' **of** $\perp \rightarrow$ *false* | **others** $\rightarrow$ *canonic*(x') **fo fo**

Defining **type** $Nat' == x{:}Nat_\perp$ **where** *canonic*(x) **convex**
we get the following non-trivial generator redefinition, renamed for perspicuity:
func $\mathbf{S}'$: $Nat' \longrightarrow Nat'$
def $\mathbf{S}'x$ == **case** x **of** $\perp \rightarrow \perp$ | **others** $\rightarrow \mathbf{S}x$ **fo qua** Nat'

Proof obligation: *canonic*$(x) \Rightarrow$ *canonic*$(\mathbf{S}'x)$. But $\perp$ is canonical and so is $\mathbf{S}x$ for canonical x different from $\perp$.

Consider the operator $\hat{} \leq \hat{}$ as defined in example 6. The following extended set of rewrite rules will be generated for Nat':

R1: $0 \leq y == true$
R2: $\mathbf{S}x \leq 0 == false$
R3: $\mathbf{S}x \leq \mathbf{S}y == x \leq y$
R4: $\mathbf{S}x \leq \bot == \bot$
R5: $\bot \leq y == \bot$
R6: $\mathbf{S}'\bot == \bot$
R7: $\mathbf{S}'0 == \mathbf{S}0$
R8: $\mathbf{S}'\mathbf{S}x == \mathbf{S}\mathbf{S}x$

R1-3 follow from the user definition, whereas R4-8 are added behind the scenes. Notice that the expression $\mathbf{S}'\bot \leq 0$ can only simplify to $\bot$, using R6 followed by R5. That agrees with the fact that the operator is strict in its first argument. The expression $\mathbf{S}\bot \leq 0$ on the other hand would simplify wrongly to *false*; however, it is not canonical, and since the user is not authorized to apply the original generator, the expression cannot appear.

Some reasoning power is lost as the result of our construction. For instance, in the example the expression $\mathbf{S}'x \leq \mathbf{S}'y$ is irreducible, whereas the forbidden $\mathbf{S}x \leq \mathbf{S}y$ would simplify to $x \leq y$. Now, most variables occurring in an ABEL expression represent well-defined values; the only exceptions are those introduced in the left hand side of a function definition. We may thus improve the reasoning power while retaining convergence by adding rules of the form $g'(\ldots,\xi,\ldots) == g(\ldots,\xi,\ldots)$, where ξ can only be instantiated to *"ordinary" variables* (or indeed to any expression syntactically identifiable as being well-defined). Then, for ordinary variables x and y, $\mathbf{S}'x \leq \mathbf{S}'y$ would be reducible to $\mathbf{S}x \leq \mathbf{S}y$ and to $x \leq y$.

In [LO93] another approach to strictness control in term rewriting is presented, based on the construction of definedness predicates and modification of the given TGI rules. This in general leads to more complicated rule sets, but it may follow from a syntactic analysis that some of the original rules do respect strong equality. In fact, that is the case with R3 of the example, which implies that the rule $\mathbf{S}'x \leq \mathbf{S}'y == x \leq y$ can be added to our set without causing loss of convergence.

5 A comparison with other languages

When defining the TGI fragment of ABEL we have searched for language constructs supporting constructivity and the reduction of consistency proofs to syntactic checks. The syntactic restrictions do complicate the language, compared to more "liberal" ones such as OBJ and the Larch Shared Language, but we believe they are of great importance for practical program reasoning.

For a large number of common computer science examples (such as stacks, binary trees, lists, multilevel lists, etc.) the syntactic restrictions enforced by our language do not make the specifications much different from those presented in OBJ and Larch. In contrast to these languages, all kinds of putting together modules have the same semantics; and any associated proof obligations are expressed in terms of explicit first order formulas.

The PVS language is also designed to be constructive in part, [OSR93, SOR93]. As

in the TGI fragment, recursion (within function definitions) must terminate. However, whereas proof of termination in the TGI fragment is oriented towards syntactic checks, PVS causes proof obligations. In trying to keep program reasoning simple, PVS avoids partial functions and explicit errors, using semantic subtypes in function domains when needed. For instance the subtraction operator of example 4 should have the domain $\{x, y : Nat \times Nat \mid y \leq x\}$. This means, however, that reasoning is limited to provably well-defined expressions, and explicit reasoning about errors is impossible. It also leads to redundant and possibly confusing subterms in function definitions using case branching (for instance in example 4 the occurrence of $\perp$ must be replaced by an irrelevant term).

VDM and PVS may use semantic subtypes to define a one-one generator basis for data types such as sets, [JM94], for instance by restricting the domain of the *add* generator to be $\{s, x : Set \times T \mid \textbf{case } s \textbf{ of } \emptyset \rightarrow true \mid add(s, y) \rightarrow y < x \textbf{ fo}\}$ provided $<$ is a total order on T. This gives a concise definition; however, the *add* generator is partial and does not capture the traditional add operator on sets. There are no proof obligations ensuring that the defined type in fact captures the set concept!

In contrast, the ABEL approach above gives a total *add* operator with the usual semantics, and the generated proof obligations form a constructive decomposition of the desired proof burden, namely ensuring that all finite sets can be generated, and that the equality defined is the desired one. One may argue that the domain restriction on the basic *add* is valuable in itself (for instance to improve the induction rule). In the above TGI approach, this domain predicate is implicit in the canonicity predicate.

In the PVS approach of strong typing user defined functions are assumed to be strict to be strict and the boolean operators to be left-strict. The kind of non-strictness allowed in ABEL above would cause severe problems.

In the framework of order sorted algebra one may formalize both weak and strong typing by means of error supersorts and stratification, [SNG89]. Generator strictness can be specified; in fact, strictness is normally assumed for all OBJ functions. In contrast to the ABEL framework, variables in user defined rules (such as those generated from TGI definitions) range over defined values only, thereby limiting the amount of permitted rewrites. For instance the rule $0 \leq y == true$ from example 9 would require y to be well-defined (when generated by stratification), whereas the ABEL method does not. Thus, the ABEL notion of weakly correct rewriting (respecting $\sqsubseteq$) provides useful reasoning about partly well-defined terms, irreducible in OBJ.

It would be easy to enrich the TGI fragment of ABEL by mechanisms for definedness control, partly syntactic.

Acknowledgments
The authors are indebted to many colleagues for fruitful discussions and feedback, and in particular to Tore Jahn Bastiansen, Bjørn Kristoffersen, and Anne Salvesen. In addition Friedrich von Henke has given useful feedback.

References

[Ba95] T.J. Bastiansen: "Parametric Subtypes in ABEL." Research report 207, Dept. of

Informatics, University of Oslo, 1995.

[BDK95] T.J. Bastiansen, O.-J. Dahl, B. Kristoffersen: "Modules and Module Composition in ABEL." To appear.

[Da77] O.-J. Dahl: Can Program Proving be Made Practical? In *Les Fondements de la Programmation*, M. Amirchahy and D. Néel, Ed., INRIA, 1977

[Da92] O.-J. Dahl: *Verifiable Programming*. Prentice Hall International 1992.

[DO91] O.-J. Dahl, O. Owe: Formal Development with ABEL. In S. Prehn, W.J. Toetenel (Eds.): *VDM'91, Formal Software Development Methods, LNCS 552*, Springer 1991, pp. 320-362.

[DLO86] O.-J. Dahl, D.F. Langmyhr, O. Owe: "Preliminary Report on the Specification and Programming Language ABEL." Research Report 106, Dept. of Informatics, University of Oslo, 1986.

[DMN71] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard: "Simula 67 Common Base Language", NCC pub. S-22, Norwegian Computing Center, 1971.

[EO93] M. Elvang-Gøransson, O. Owe: "A simple sequent calculus for partial functions" *Theoretical Computer Science*, vol. 114, 1993, pp. 317-330.

[FGJ85] K. Futatsugi, 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.

[Gu75] J.V. Guttag: "The Specification and Application to Programming of Abstract Data Types." Ph. D. Thesis, Computer Science Department, University of Toronto, 1975.

[GHW85] J.V. Guttag, J.J. Horning, J.M. Wing: "Larch in Five Easy Pieces." Digital Systems Research Center, Palo Alto, California, July 1985.

[JM94] C.B. Jones, C.A. Middelburg: "A Typed Logic of Partial Functions Reconstructed Classically." *Acta Informatica* 31: 399-430, 1994.

[KD95] B. Kristoffersen, O.-J. Dahl: On Introducing Higher Order Functions in ABEL. In preparation.

[LO93] O. Lysne, O. Owe: "Definedness and Strictness in Generator Inductive Definitions." Part of Lysne's dr. scient. thesis, Univ. of Oslo, Dec. 1991.

[Mi83] R. Milner: "A Proposal for Standard ML." Report CSR-157-83, Computer Science Dept., Edinburgh University, 1983.

[Na83] R. Nakajima, T. Yuasa (Eds.): *The IOTA Programming System, LNCS 160*. Springer 1983.

[OD91] O. Owe, O.-J. Dahl: "Generator Induction in Order Sorted Algebras." *Formal Aspects of Computing*, 3:2-20, 1991

[Ow93] O. Owe: "Partial Logics Reconsidered: A Conservative Approach" *Formal Aspects of Computing* (Springer Verlag), vol. 5, 1993, pp. 208-223.

[OSR93] S. Owre, N. Shankar, J.M. Rushby: "The PVS Specification Language." Computer Science Lab., SRI International, Menlo Park, CA, 1993.

[SOR93] N. Shankar, S. Owre, J.M. Rushby: "A Tutorial on Specification and Verification Using PVS." Computer Science Lab., SRI International, Menlo Park, CA, 1993.

[SNG89] G. Smolka, W. Nutt, J.A. Goguen, J. Meseguer: "Order-sorted equational computation", in *Resolution of Equations in Algebraic Structures, Vol. II: Rewriting Techniques*, Academic Press, 1989, pp. 299-369.