Nordic Journal of Computing 5(1998), 1-31

Subtyping and Constructive Specification

Ole-Johan Dahl Olaf Owe
Tore J. Bastiansen
Department of Informatics, University of Oslo, Norway
{olejohan,olaf,toreba}@ifi.uio.no

Abstract. The use of subtyping is explored in the context of a style of constructive specification. The intention is to enhance expressiveness and strengthen syntactic controls. The resulting specification style is part of a wide spectrum language, ABEL, developed at the Oslo University. A concept of signature completeness is presented, as well as a completion algorithm. Some results relating to weak, strong, and optimal typing are proved.

CR Classification: D.1.1, D.2.1, D.2.4, D.3.3, F.3.1, F.3.3.

Key words: Algebraic specification, Specification language, Generator induction, Subtypes, Partial functions, Definedness, Syntactic control.

1. Introduction

Specifications are used to express system requirements and system design at different levels of abstraction. Ultimately specifications will function as correctness criteria for the implemented system. Formal techniques, expressing system properties by axioms, say in a typed first order formula language, may allow mechanized aids to reasoning about the specifications themselves, as well as aids to program development and verification with respect to the specifications.

Specifications should be easy to understand, analyze, and to experiment with. Good modularization is essential for more detailed system specification and implementation. Even so, using free axiomatic techniques may lead to specifications which are internally inconsistent and therefore useless. Consistency proofs are difficult in general, so there is a need for linguistic aids for avoiding inconsistencies, or at least detecting inconsistency at an early stage.

We believe that *constructive* specifications represent important intermediate stages between property oriented specifications and implementation, for the following reasons:

(1) It is possible to provide better syntactic control with consistency (as well as completeness).

(2) Verifying that all property requirements are fulfilled by a constructive specification is a way of proving the consistency of the former.

Received November 3, 1995, revised December 9, 1996; accepted December 8, 1997.

(3) A constructive specification is an "abstract", but executable, program which may function as a prototype system (possibly inefficient).

It is clear that requirements specifications will normally be expressed by means of free axioms. However, if some of the types and functions involved are constructive, then the axioms may well be easier to understand, and more efficient and powerful mechanized tools for reasoning about the specifications are likely to be available. An example is the requirement specification in [5] of a many-lift system whose internal consistency is almost trivial.

We shall show how subtypes may be used to enhance expressiveness and strengthen syntactic controls in the context of a certain style of constructive specification. More specifically subtyping is a means to:

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

Although the results of this paper to a large extent are language independent, we need a language in which to express our ideas. For that purpose we use the applicative core of a specification and programming language called *ABEL*, (Abstraction Building, Experimental Language), [6, 5, 4], developed at the University of Oslo, and based on a typed logic for partial functions, [21, 8]. A separate paper, [16], will discuss the extension to higher order. The most important sources of ideas have been as follows: SIMULA 67 (classes and subclasses), [7], the LARCH and IOTA activities (generator induction), [11, 12], [19], and OBJ (order sorted algebras), [9, 10].

The so called *TGI fragment* of ABEL deals with constructively defined types and functions (both partial and total). TGI stands for "Terminating Generator Induction". TGI specifications give rise to convergent sets of rewrite rules, which enable efficient manipulation of expressions for purposes of simplification and proof. Two species of subtypes are considered: *syntactic subtypes* defined inductively, and *semantic subtypes* defined by means of predicates. A concept of *convex* subtype is identified, useful for the strengthening of expression typing. An important property of the TGI fragment is that all consistency related proof obligations can be identified and formulated mechanically (as formulas in typed first order logic). We also believe that the syntactic TGI restrictions will provide useful guidance for programmers who are not trained algebraists.

The use of syntactic TGI-like restrictions have also been advocated by Turner for educational purposes [25].

The paper is organized as follows. In Chapter 2 the TGI fragment is described and exemplified. Chapter 3 deals with subtypes and weak typing. Chapter 4 specifically deals with parameterized syntactic subtypes, as well

as strong and optimal typing. An algorithm for "signature completion" is presented and proved. A complete signature in principle consists of maximally strong profiles for all possible (syntactic) subdomains of each defined function. In Chapter 4 semantic subtypes are treated. Two nontrivial examples of their use are given.

2. TGI Specifications without Subtypes

The TGI language 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. (A pilot implementation of parts of ABEL exists; in particular, most of the algorithms presented in the paper are implemented.)

A type T defines a pair

$$T \stackrel{\text{def}}{=} (\mathcal{V}_T, \mathcal{F}_T)$$

where $\mathcal{V}_T$ is the set of values of type T, and $\mathcal{F}_T$ 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 $T_1 \times T_2 \times \ldots \times T_{n_f}$ and its codomain U, where $n_f \geq 0$ is called the arity of f. Note that constants are functions with zero arity.

The set of profiles for the declared functions in an ABEL specification is called the *signature* of the specification.

The value set of a type T is constructively 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 $\mathcal{V}_T$. 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 [18]). In order that the representation set be non-empty, at least one generator must have no T-argument.

For instance, functions $0 :\longrightarrow Nat$ and $\mathbf{S}\hat{}\colon Nat \longrightarrow Nat$ may be taken as generators for the type Nat of natural numbers. Then the ground generator terms of type Nat are: $0, \mathbf{S}0, \mathbf{SS}0, \mathbf{SSS}0 \ldots$, which, in our minds would correspond to the natural numbers $0, 1, 2, 3, \ldots$.

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".

Example: the function $\hat{}+\hat{}: Nat \times Nat \longrightarrow Nat$ may be TGI defined by induction on the second argument, thus:

$$\textbf{def}\, x + y == \textbf{case}\, y\, \textbf{of}\, 0 \to x \mid \mathbf{S}z \to \mathbf{S}(x + z)\, \textbf{fo}$$

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. It is easy to provide fairly sophisticated syntactic checks; for instance, the lexicographic extension (left to right) of the subterm check is sufficient to prove the termination of the following TGI definition of the well known Ackermann function:

$$\begin{array}{ll}\textbf{func} & Ack\colon Nat \times Nat \longrightarrow Nat \\ \textbf{def} & Ack(x, y) == \textbf{case}\, x\, \textbf{of}\, 0 \to \mathbf{S}y \mid \mathbf{S}x' \to \\ & \quad \textbf{case}\, y\, \textbf{of}\, 0 \to Ack(x', \mathbf{S}0) \mid \mathbf{S}y' \to Ack(x', Ack(x, y'))\, \textbf{fo fo}\end{array}$$

Even so, ad-hoc termination proofs may sometimes have to be provided (otherwise the "halting problem" of algorithm theory would be solvable). The termination of the function definitions occurring in the present paper is easily checked syntactically.

The form of TGI function definitions ensures logical consistency as well as ground completeness (with some reservations for the treatment of equality, see below). This follows from syntactic checks for non-overlapping and exhaustive discriminators 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. For instance:

$$\begin{array}{ll} x + 0 & == x \\ x + \mathbf{S}y & == \mathbf{S}(x + y)\end{array}$$

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 useful reasoning aid within the TGI framework.

Note that the TGI fragment 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):

⟨type definition⟩ ::= **type** ⟨type identifier⟩⟨formal param part⟩$^?$
== ⟨type module⟩
⟨formal param part⟩ ::= {⟨type identifier list⟩}
⟨type module⟩ ::= **module** ⟨type module item list⟩
endmodule

The items of a type module include profiles and definitions of functions, as well as a generator basis specification. The set $\mathcal{F}_T$ of a type T consists of those functions which are introduced in its type module (the grouping of function symbols by type modules and modules of other kinds plays a role for function overloading, cf. [5]). The following items are implied, even for a formal type T (including strict and non-strict equality, see below):

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

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

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

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

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). Thus, for the type *Nat* the rule is the usual one for mathematical induction:

$$\frac{\vdash P(0);\ P(x) \vdash P(\mathbf{S}x)}{\vdash \forall x{:}\mathit{Nat} \bullet P(x)}$$

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:

$$\begin{aligned}\textbf{def}\, x{=}y == \textbf{case}\,(x,y)\,\textbf{of}\ &|_{i=1}^{n}\ (g_i(\bar{x}_i), g_i(\bar{y}_i)) \to \bar{x}_i{=}\bar{y}_i\\ &\mid \textbf{others} \to \mathit{false}\ \textbf{fo}\end{aligned}$$

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} ==
module      - - finite sequences of values of an unspecified type
  func ε: ⟶ Seq                                  - - empty sequence
  func ^⊢^: Seq × T ⟶ Seq                        - - append right
  func ^⊣^: T × Seq ⟶ Seq                        - - append left
  func ^⊢⊣^: Seq × Seq ⟶ Seq                     - - concatenate
  one-one genbas ε, ^⊢^                  - - a traditional choice, [3]
  def  x ⊣ q == case q of ε → ε ⊢ x | q' ⊢ y → (x ⊣ q') ⊢ y fo
  def  q ⊢⊣ r == case r of ε → q | r' ⊢ x → (q ⊢⊣ r') ⊢ 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:

$$\begin{aligned}\textbf{def}\ q{=}r == \textbf{case}\,(q,r)\,\textbf{of}\ &(\varepsilon,\varepsilon) \to \mathit{true}\\ &\mid (q'{\vdash}x, r'{\vdash}y) \to q'{=}r' \wedge x{=}y\\ &\mid \textbf{others} \to \mathit{false}\ \textbf{fo}\end{aligned}$$

Strictly speaking $\mathit{Seq}\{T\}$, where T is a formal parameter, is not a type, but a *type schema*. Type schemata give rise to types through instantiation. Thus, $\mathit{Seq}\{\mathit{Nat}\}$ and $\mathit{Seq}\{\mathit{Seq}\{\mathit{Bool}\}\}$ are types obtained from the *Seq* schema. However, by looking at formal type parameters as types not constructively specified, the distinction between types and type schemata for many purposes disappears.

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 the contents are
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 (total and strict):

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

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. For that reason there is an associated obligation to prove for any function f with an argument inductive over T that the substitution property holds for that argument:

$$x \stackrel{T}{=} y \Rightarrow f(\ldots, x, \ldots) = f(\ldots, y, \ldots)$$

(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. Thus, from a "concrete" point of view, say that of a term rewriting machine, TGI defined functions are total, and the symbol $\perp$ is a member of the value set of every type (cf. Section 4.2). From the intended more "abstract" point of view, functions are potentially partial, and function applications potentially *ill-defined*, having no value. From this viewpoint the symbol $\perp$ may be thought of as an infinite computation.

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 strict equality, the non-strict "strong" equality can be implemented constructively behind the scenes. Since strong equality is non-monotonic it is, however, not allowed in the right-hand sides of TGI function definitions. Notice, however, that $==$ is the main operator of function definitions.

Generators are strict by definition, $g(\ldots, \perp, \ldots) == \perp$. Furthermore, **case** constructs are strict in the discriminand, **case** $\perp$ **of** $\ldots$ **fo** $== \perp$. The latter implies that only functions monotonic with respect to definedness can be user defined.

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: $t \sqsubseteq u$. (The operator $\sqsubseteq$, approximation with respect to well-definedness, is not part of the TGI language fragment.) 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 4.2.

3. Subtypes

We introduce the following (meta) relations on types, including type schemata: $=$ (equal), $\prec$ (subtype), and $\preceq$ (subtype or equal). The equality relation represents name 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 types T and U meaningful in the context of any ABEL specification (no proof of this fact will be given here):

$$\text{SUBTY:}\ \ T \preceq U \ \Rightarrow\ \mathcal{V}_T \subseteq \mathcal{V}_U \ \wedge\ \mathcal{F}_U \subseteq \mathcal{F}_T$$

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.

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

$$\text{EMPTY:}\ \ \mathcal{V}_\oslash \triangleq \emptyset \ \wedge\ \mathcal{F}_\oslash \triangleq \langle\text{all declared functions}\rangle$$

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

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

The subtype relation only applies to constructively defined types, and the language syntax ensures that each non-empty type has a unique maximal supertype:

$$\forall T \bullet T \neq \oslash \Rightarrow \exists U \bullet T \preceq U \wedge \forall V \bullet (T \preceq V \Rightarrow V \preceq U)$$

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. For any given ABEL specification, the set of declared types, ordered by the subtype relation, has the property that the smallest common supertype of related types is unique, and so is the largest common subtype (always defined). These concepts are expressed using the meta-operators $\sqcup$ and $\sqcap$, respectively.

Types T, U whose only common subtype is $\oslash$ are said to be *disjoint*, $T \precsim U$.

$$T \precsim U \ \triangleq\ T \sqcap U = \oslash$$

Unrelated types are disjoint by definition. EMPTY implies that disjoint types have disjoint value sets. (The converse does not hold.)

As we shall see, the operators $\preceq$, $\sqcup$, and $\sqcap$ are syntactically defined for parameterless types and type schemata. All three are extended pointwise to type products. This is a special case of their generalization to parameterized types. The following definitions, where X is a list of formal type parameters, and $\circ$ stands for either of $\sqcup$ and $\sqcap$, are in agreement with the SUBTY principle:

$$\begin{aligned} U\{T_1, \ldots, T_n\} \preceq U'\{T'_1, \ldots, T'_n\} \ &\triangleq \\ &T_1 \preceq T'_1 \wedge \ldots \wedge T_n \preceq T'_n \wedge U\{X\} \preceq U\{X\} \\ U\{T_1, \ldots, T_n\} \circ U'\{T'_1, \ldots, T'_n\} \ &\triangleq\ U''\{T_1 \circ T'_1, \ldots, T_n \circ T'_n\}, \\ &\text{where}\ \ U''\{X\} = U\{X\} \circ U'\{X\} \end{aligned}$$

The following properties hold:

$\mathcal{V}_{T \sqcup U} = \mathcal{V}_T \cup \mathcal{V}_U$	for related parameterless types and type schemata (interpreting formal types as the type *Nat*),
$\mathcal{V}_T \cup \mathcal{V}_U \subseteq \mathcal{V}_{T \sqcup U}$	for related types generally,
$\mathcal{V}_{T \sqcap U} = \mathcal{V}_T \cap \mathcal{V}_U$	always, but see Section 3.2.

An important monotonicity principle for parameterized types follows from the generalized subtype relation:

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

Assume $T \preceq U$. The fact that any T value belongs to $\mathcal{V}_U$ gives rise to the following typing rules for expressions:

- A *well-typed* expression of *minimal type* T is either:
  - $\perp$ if $T = \oslash$, or
  - 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 each 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, or
  - A **case** (or **if**) construct whose relevant 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 relevant alternatives are those whose discriminator types are not disjoint with the minimal type of the discriminand. (A reoccurrence of a **case** discriminand of type U within an alternative headed by a discriminator of type U' has the type $U \sqcap U'$.)

> REMARK: In the context of subtypes it may be useful to allow 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 signature concept is extended accordingly. The profile sets occurring in the TGI fragment of ABEL are *weakly regular*, cf [20]. This means that, although several profiles for a function f may fit an application of f, the minimal codomain of such profiles is unique.

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. Conversely, a signature is said to be *sound* if each function definition is well-typed with respect to every profile for that function.

The fact that $\oslash$ is included in all types has the consequence that any function profile specifies a (possibly) partial function. Since $\bot$ 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 contains applications of total functions only, and does not contain $\bot$.

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

> THEOREM 1.
> *Let e be a well-typed expression of minimal type T with respect to a sound signature. Then any ground instance of e either has a value of a type included in 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*.

> 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 may 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 the type of e to become T_1. ABEL provides two alternative coercion mechanisms:

- "Proof time coercion", $e\,\mathbf{qua}\,T_1$, 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", $e\,\mathbf{as}\,T_1$, 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 is specified as follows:

  $\mathbf{func}\ \hat{}\,\mathbf{as}\,T_1 :\ T \longrightarrow T_1$
  $\mathbf{def}\quad x\,\mathbf{as}\,T_1 == \mathbf{if}\,x\,\mathbf{in}\,T_1\,\mathbf{th}\,x\,\mathbf{el}\,\bot\,\mathbf{fi}$ - - type correct by definition

  where the predicate $\hat{}\,\mathbf{in}\,T_1 : T \longrightarrow Bool$ may be defined automatically, as explained later.

Clearly $e\,\mathbf{qua}\,T_1 == e$ and $e\,\mathbf{as}\,T_1 \sqsubseteq e$. The minimal type of both coercions is $T_e \sqcap T_1$, where T_e is the minimal type of e. It is being debated whether or not coercion insertion of some kind should be automatic. In view of the typing rules 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

For ease of explanation we introduce the concept of syntactic subtypes through an example showing a syntactic type hierarchy:

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 number is positive
         N^: Pos ⟶ Neg
              - - the negation of a positive number is negative
    one-one genbas 0, S^, N^
    endmodule
```

The types listed following the keyword **by** are called *basic subtypes*. They are by definition disjoint. The *main type* is the union of its basic subtypes, $Int = Zero \sqcup Pos \sqcup Neg$. Each basic type must occur as the codomain of at least one generator. For a one-to-one generator basis the disjointness is a direct consequence of the one-one property; for a many-to-one basis, however, proofs of disjointness are required.

Intermediate syntactic subtypes may be defined as unions of subsets of basic subtypes, as exemplified by the list of definitions following the keyword **with**.

In the example, the domains of the unary generators are subtypes chosen so that the well-typed generator terms are in a one-to-one correspondence with the integers. For the basic subtypes we have:

$$\mathcal{V}_{Zero} = \{0\}, \quad \mathcal{V}_{Pos} = \{\mathbf{S}0, \mathbf{SS}0, \ldots\}, \quad \mathcal{V}_{Neg} = \{\mathbf{NS}0, \mathbf{NSS}0, \ldots\}.$$

The intermediate types *Nat*, *NZro* and *NPos* represent the indicated type unions. *Nat*, as well as *Int* and *Zero*, are convex types. This follows from the generator profiles.

For the rest of the paper we shall assume that the codomains of generator profiles are mutually distinct basic types, and that all possible intermediate subtypes are defined (or are introduced implicitly). It follows that a syntactic type hierarchy is a lattice whose maximal and minimal elements are the main type and $\oslash$, respectively, and that each member may be identified with the set of basic subtypes it contains. The lattice operations correspond to union and intersection of these sets, and, since basic subtypes are value disjoint, to the same operations on the corresponding value sets.

An **in**-predicate for a syntactic subtype is defined by a **case** expression in the obvious way.

3.2 Parameterization

For a parameterized family of syntactic subtypes the parameter list associated with a subtype may be a sublist of that of the main type. For instance, the subtype *ESeq* of the following example has an empty parameter list.

EXAMPLE 6.
We define sequence types as the union of two basic subtypes, representing empty and nonempty sequences, respectively:

type $Seq\{T\}$ **by** $ESeq, NESeq$ ==
module
  func ε: $\longrightarrow ESeq$, $\hat{}\vdash\hat{}$: $Seq \times T \longrightarrow NESeq$
  one-one genbas ε, $\hat{}\vdash\hat{}$
    – Some functions apply to nonempty sequences only:
  func rt: $NESeq \longrightarrow T$ - - right term
  def $rt(q\vdash x) == x$
  func lr: $NESeq \longrightarrow Seq$ - - left rest
  def $lr(q\vdash x) == q$
  - - - - - - -
endmodule

One might wish to split the type *NESeq* into "singleton sequences", *SgSeq*, and "long sequences", *LgSeq*. This is achieved by identifying the new subtypes as basic (replacing *NESeq*), defining *NESeq* as their union, and replacing the $\hat{}\vdash\hat{}$ profile by the following two:

$$\hat{}\vdash\hat{}: ESeq \times T \longrightarrow SgSeq$$
$$\hat{}\vdash\hat{}: NESeq \times T \longrightarrow LgSeq$$

The fact that the generator ε of the example is a constant implies that the subtype *ESeq* must be parameterless; otherwise the expression ε would not have a unique minimal type. It follows that *ESeq* is a syntactic subtype of any instance of $Seq\{T\}$. For $Seq\{\oslash\}$, however, we find that all generator

terms except ε are ill-defined due to generator strictness. Thus, we wish $Seq\{\oslash\} = ESeq$ to hold by definition (in agreement with MONOTY).

The generator profiles implicitly determine the parameterization of each syntactic subtype. In general a simple fixpoint algorithm is needed, starting with lists of formal types occurring explicitly in the domain of some generator of each type, and then adding implicit ones iteratively. In this way the parameterization of the main type is also checked. It is perhaps a question of taste whether or not the parameterization of each subtype should be indicated explicitly by the user. If not, the system should respond with the computed parameter lists.

As a means of strengthening the typing algorithm we investigate the conditions under which a "monotonicity" property for disjointness can be deduced:

$$U \mathrel{\text{-}\!\!\diamond\!\!\text{-}} V \Rightarrow T\{\ldots, U, \ldots\} \mathrel{\text{-}\!\!\diamond\!\!\text{-}} T\{\ldots, V, \ldots\}$$

If so, the formal parameter in question is said to be *disjointness preserving*. A sufficient condition is that all T values, i.e. ground T generator terms, contain subterms of that parameter type, and that the generator basis of T is one-to-one.

We may determine the set $\mathcal{DP}(T)$ of (indices of) disjointness preserving parameters for each member T of a hierarchy of syntactic subtype schemata on the basis of the generator set. Define the function $\mathcal{L}$: ⟨type (schema)⟩ $\longrightarrow$ ⟨formal parameter set⟩ as follows:

- For a hierarchy member T: $\mathcal{L}(T)$ initially is the set of formal parameters parameterizing T, as determined above,
- For a formal parameter X: $\mathcal{L}(X) \triangleq \{X\}$,
- For any other type expression: $\mathcal{L}(U\{T_1, \ldots, T_n\}) \triangleq \bigcup_{i \in \mathcal{DP}(U)} \mathcal{L}(T_i)$, where $n \geq 0$.

Consider the following fixpoint iteration:

(1) Choose some generator in the hierarchy, $T_1 \times \ldots \times T_n \longrightarrow U$, $n \geq 0$.

(2) Let $M = \bigcup_{i=1}^{n} \mathcal{L}(T_i)$.

(3) For each $V \succeq U$ replace $\mathcal{L}(V)$ by $\mathcal{L}(V) \cap M$.

In the (maximal) fixpoint $\mathcal{L}(T) = \mathcal{DP}(T)$ for each member of the hierarchy.

Let T be a convex type schema. Then a type expression of the form $T\{\ldots, \oslash, \ldots, \oslash, \ldots\}$ must be equal to the union of basic T subtypes not parameterized by the $\oslash$'s, provided that these parameters are disjointness preserving for all other basic T subtypes. Referring to Example 6, it now follows by purely syntactic reasoning that $Seq\{\oslash\} = ESeq$, using the fact that the parameter of the other basic subtype(s) is disjointness preserving.

For a many-to-one generator basis the situation is rather more complicated in general. It is conceivable that value components of a parameter type contained in all T generator terms could be ignored in the equality relation

for T. If so, the types $T\{\ldots,U,\ldots\}$ and $T\{\ldots,V,\ldots\}$ could be disjoint syntactically, but not semantically.

Example: Define a type schema $Set'\{T\}$ identical to $Set\{T\}$ of Example 3, except that the generator $\emptyset$ has the profile $T \longrightarrow Set$, where the parameter of $\emptyset(x)$ is never accessed in function definitions. Then $Set'\{Pos\} \sqcap Set'\{Neg\} = Set'\{Pos \sqcap Neg\} = Set'\{\oslash\} = \oslash$, where the last equality follows from generator strictness. But $\mathcal{V}_{Set'\{Pos\}}$ and $\mathcal{V}_{Set'\{Neg\}}$ would have a common element, represented by ground instances of $\emptyset(x)$ for $x : NZro$, all semantically equal.

In order to avoid such anomalies it is sufficient to restrict the ABEL language by forbidding generator arguments of parameter types, which are redundant with respect to the equality relation over the parameterized type.

3.3 Signature completion

TGI function definitions in the context of syntactic subtypes may be subjected to a type analysis which goes deeper than just checking the validity of user submitted function profiles. Let $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ be the total set of functions (including generators) occurring in an ABEL specification. The signature for $\mathcal{F}$ contains one profile provided by the user, say $f_i : D_i \longrightarrow C_i$ for each function, where D_i is a type product with zero or more components. It defines the largest domain on which f_i can legally be applied.

As already mentioned (see the remark on page 10) we shall permit additional profiles in the signature in order to provide for stronger typing. We now seek a "best possible" signature for $\mathcal{F}$, i.e. one which will provide the strongest possible (weak) typing of expressions using our typing algorithm. It is possible that such a signature would have to contain profiles $f_i : D_{ij} \longrightarrow C_{ij}$ for every domain included in D_i, for each non-generator function f_i (including domains containing $\oslash$). In addition it should contain the generator profiles provided by the user, as well as profiles expressing generator strictness wrt. each individual argument. Such a signature is said to be *full for* $\mathcal{F}$. The inclusion of $\oslash$ as a "basic" subtype at this point will provide for function strictness analysis.

Let $\mathcal{S}$ be a weakly regular signature full for $\mathcal{F}$. We construct a signature $\mathcal{S}'$ full for $\mathcal{F}$ as follows: For every non-generator profile $f : D \longrightarrow C$ in $\mathcal{S}$ compute the minimal type U' of an application $f(\bar{x})$ with $\bar{x} : D$ by applying the typing algorithm to the RHS of the definition of f wrt. the context $\mathcal{S}$. $f : T \longrightarrow U'$ is the corresponding profile of $\mathcal{S}'$. It can be proved that $\mathcal{S}'$ is weakly regular. Since $\mathcal{S}'$ is unique there is a function Φ such that $\mathcal{S}' = \Phi(\mathcal{S})$. Any fixpoint of Φ is semantically correct in the sense that it respects the given function definitions.

It follows from the typing algorithm that Φ is monotonic with respect to the following order of signatures full for $\mathcal{F}$:

> $\mathcal{S}$ is less than or equal to $\mathcal{S}'$ iff $C_{ij} \preceq C'_{ij}$
> for every i, j such that $D_{ij} \longrightarrow C_{ij} \in \mathcal{S}$ and $D_{ij} \longrightarrow C'_{ij} \in \mathcal{S}'$.

Clearly the desired signature for $\mathcal{F}$ is the *minimal* fixpoint $\mathcal{S}^*$ of Φ. Since the set of subtypes of user defined codomains is finite, $\mathcal{S}^*$ can be obtained

by a finite number of applications of Φ, starting with the minimal signature $\mathcal{S}_0$ in which all non-generator profiles have the codomain $\oslash$. $\mathcal{S}_0$ is weakly regular.

We call $\mathcal{S}^*$ the *complete* signature for $\mathcal{F}$ in order to indicate that it contains all type information that can be extracted from the set of function definitions by syntactic means.

Let the profiles $T \longrightarrow U$ and $T' \longrightarrow U'$ in $\mathcal{S}^*$ be associated with the same function. If $T \prec T'$ it is easy to show that $U \preceq U'$ must follow. If $U = U'$ then the former profile is *redundant* wrt. the typing algorithm. It follows that the complete signature is adequately represented by the (unique) subset of $\mathcal{S}^*$ obtained by deleting all redundant profiles.

The signature completion algorithm can be decomposed by considering one defined function at a time, in an order determined by the function dependency graph. Mutually recursive functions, however, must be treated simultaneously.

Further speeding up is possible using the fact that parameterless types and type schemata have the following "splitting" property:

$$\bigcup_{T' \in \mathcal{B}(T)} \mathcal{V}_{T'} = \mathcal{V}_T, \text{ where } \mathcal{B}(T) \text{ is the set of basic subtypes of } T.$$

The result generalizes easily to type products. (Definedness monotonicity implies $\mathcal{B}(T)$ need not contain $\oslash$.) Assume for the moment that the function set $\mathcal{F}$ is defined over parameterless types and type schemata only. Then the codomain of a profile $f : T \longrightarrow U$ on any non-basic domain may be computed by joining the codomains of a set of profiles on basic domains:

$$U = \bigsqcup \{U' \mid f : T' \longrightarrow U' \in \mathcal{S}^* \wedge T' \in \mathcal{B}(T)\}$$

Consequently it is sufficient to carry out the completion algorithm on sets of basic profiles (including domains containing $\oslash$ if strictness analysis is desired).

EXAMPLE 7.
Let *Bool* have the basic subtypes *False* and *True*, where $\mathcal{V}_{False} = \{false\}$ and $\mathcal{V}_{True} = \{true\}$, 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 completion algorithm for $\hat{} \leq \hat{}$ works as follows, dealing with profile sets $\mathcal{P}_0, \mathcal{P}_1, \ldots$:

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

Since $\mathcal{P}_2 = \mathcal{P}_3$, this is the minimal fixpoint. The profile $\mathit{Zero} \times \oslash \longrightarrow \mathit{True}$ indicates that $\hat{} \leq \hat{}$, as defined above, is non-strict in its second argument. Notice the change from $\oslash$ to *Bool* in the codomain of the last profile in the list. The codomain is the type of the RHS expression $x' \leq y'$ where, according to the profile of the generator $\mathbf{S}\hat{}$, both variables are of the type *Nat*. Consequently the codomain in $\mathcal{P}_2$ must be the type union of all the codomains in $\mathcal{P}_1$ (omitting profiles with domain containing $\oslash$).

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

$\oslash \times \mathit{Nat} \longrightarrow \oslash$
$\mathit{Zero} \times \mathit{Nat} \longrightarrow \mathit{True}$
$\mathit{Pos} \times \oslash \longrightarrow \oslash$
$\mathit{Pos} \times \mathit{Zero} \longrightarrow \mathit{False}$
$\mathit{Nat} \times \mathit{Nat} \longrightarrow \mathit{Bool}$

Notice that the profile $\mathit{Zero} \times \oslash \longrightarrow \mathit{True}$ is included in $\mathit{Zero} \times \mathit{Nat} \longrightarrow \mathit{True}$. The totality of $\hat{} \leq \hat{}$ does not follow from the above analysis; see, however, the following subsections.

If instances of parameterized types (i.e. with non-formal parameters) occur in the $\mathcal{F}$ signature, all combinations of parameter subtypes may have to be considered. It is possible, however, to reduce the number of combinations if the "splitting" into subtypes works for parameter types. The following example, however, shows that that is not always the case: The value set $\mathcal{V}_{\mathit{NESeq}\{\mathit{Pos}\}} \cup \mathcal{V}_{\mathit{NESeq}\{\mathit{Neg}\}}$ consists of sequences where either all elements are positive or all are negative. This set is smaller than $\mathcal{V}_{\mathit{NESeq}\{\mathit{NZro}\}}$.

Intuition tells us that splitting the parameter of $T\{U\}$ works if and only if $T\{U\}$ values can contain at most one U value. This is in fact the case for type products of length n, seen as instances of a type schema with n parameters. In [1] it is shown that splitting works for the U parameter position of $T\{\ldots, U, \ldots\}$ iff no T generator contains more than one occurrence of U, textually or indirectly. A fixpoint algorithm is needed here in general.

EXAMPLE 8.
We define a function adding together a sequence of nonzero integers, where *Seq* is the extended version of the type schema of Example 6.

func $sum\colon Seq\{NZro\} \longrightarrow Int$
def $sum(q) ==$ **case** q **of** $\varepsilon \to 0 \mid q' \vdash x \to sum(q') + x$ **fo**

In order to construct a complete profile set for *sum* it is sufficient to compute the following profiles:

$$\begin{array}{ll} sum\colon & \oslash \longrightarrow \oslash \\ & ESeq \longrightarrow Zero \\ & SgSeq\{Pos\} \longrightarrow Pos \\ & SgSeq\{Neg\} \longrightarrow Neg \\ & LgSeq\{Pos\} \longrightarrow Pos \\ & LgSeq\{Neg\} \longrightarrow Neg \\ & LgSeq\{NZro\} \longrightarrow Int \end{array}$$

Notice that the profile $SgSeq\{NZro\} \longrightarrow NZro$ may be constructed as the union of the two with basic actual parameter types, whereas the last profile must be part of the fixpoint construction.

3.4 Syntactic well-definedness control

The above type-analysis may be extended to recognize well-defined terms in some cases. In order to do this syntactically, function profiles will be classified as total or partial. Generator profiles are total by definition. A total profile, say $D \longrightarrow C$ [*total*] (where D or C do not contain $\oslash$), is *valid* iff the function applied to a value of type (included in) D results in a value of type (included in) C; for ill-defined argument the result may be ill-defined or a value of type C. For instance, the partial subtraction operator on *Nat* (from Example 4) may have the profiles $Nat \times Zero \longrightarrow Nat$ [*total*], and $Nat \times Nat \longrightarrow Nat$ [*partial*]; whereas *Nat*-addition may have the profile $Nat \times Nat \longrightarrow Nat$ [*total*]. Similarly, for $T' \leq T$ the partial function $\hat{}$ **as** T' : $T \longrightarrow T'$ has the total profile $T' \longrightarrow T'$. And the **qua** mechanism may be considered a total function, $\hat{}$ **qua** $T'\colon T \longrightarrow T'$ [*total*].

A total profile may not have $\oslash$ as codomain, but it may have $\oslash$ in its domain, indicating a non-strict argument position, as in the profile $Zero \times \oslash \longrightarrow True$ [*total*] for the less-than-or-equal predicate on *Nat* (defined in Example 7), implying that the term $0 \leq \perp$ is well-defined (and of type *True*). Notice that the symbol $\perp$ is an "ill-defined constant", $\perp\colon \longrightarrow \oslash$.

A weakly regular signature is said to be *t-regular* if for any total profile for f all other f-profiles with a smaller domain are total as well, except those having $\oslash$ in both domain and codomain. A partial profile is made redundant by a total profile with equal or larger domain and equal or smaller codomain,

whereas a total profile is not made redundant by any partial profile. And a total profile with $\oslash$ in its domain is not redundant by one without $\oslash$.

The typing algorithm will select a total profile, if possible, when several profiles give the same minimal codomain. (This does not change the concept of minimal type). We say that a *function application is total* if the profile selected in the type analysis is total (and partial otherwise).

A term is *syntactically well-defined* if it is well-typed, all variables are of types other than $\oslash$, and all function profiles selected in the type-analysis are total; however, arbitrary well-typed subterms may be accepted in non-strict argument positions as follows: if a total profile is selected for arguments of type $D_1 \times T \times D_2$, and there is a total profile for the same function with domain $D_1' \times \oslash \times D_2'$, where $D_1 \times D_2 \preceq D_1' \times D_2'$, then any well-typed term of type T may be accepted in the non-strict argument position.

Examples: Let $x, y : \mathit{Nat}$. The terms $x + \mathbf{S}(y + x)$, $x \dot{-} 0$ and even $0 \leq \bot$ and $0 \leq x \dot{-} y$ are syntactically well-defined, but the terms $x + \bot$, $x \dot{-} y$ and $x \dot{-} x$ are not (although the latter is semantically well-defined). Let $\mathbf{P}\hat{}$ have the profiles $\mathit{Zero} \longrightarrow \oslash$ [*partial*], $\mathit{Nat} \longrightarrow \mathit{Nat}$ [*partial*], and $\mathit{Pos} \longrightarrow \mathit{Nat}$ [*total*]. Then the term $\mathbf{PS}x$ is syntactically well-defined (of type *Nat*), and $\mathbf{SP}x$ is of type *Pos*, but not syntactically well-defined, whereas $\mathbf{P}0$ is of type $\oslash$, and is thus recognized as ill-defined.

The concepts of well-typed function definition and sound signature are strengthened by insisting that for each total profile of a defined function, and for left-hand side variables typed according to the profile, the right-hand side must be syntactically well-defined.

Alternatively a user provided total profile $f : D \longrightarrow C$ whose totality can not be validated syntactically, could give rise to a semantic proof obligation. However, in some cases the totality may be disproved by the completion process, *viz.* if a profile $D' \longrightarrow \oslash$ is constructed for f where $D' \preceq D$ and any $\oslash$ component of D' is in D as well.

THEOREM 2.
A syntactically well-defined term is semantically well-defined (for all type correct and well-defined values of the free variables).
PROOF:
By induction on the structure on terms one may first prove the theorem for all ground terms (assuming the signature to be sound). If a term t is syntactically well-defined and all selected profiles are total, then by t-regularity any ground instance of t will be syntactically well-defined as well. If not all selected profiles are total, let t' be t with all subterms which are not syntactically well-defined, replaced by $\bot$. Now t' will be syntactically well-defined, and any ground instance of it will be syntactically well-defined as well (by t-regularity). Thus t' is semantically well-defined; and since the language for expressing right-hand sides is monotonic, t must be semantically well-defined.

The signature completion algorithm can easily be extended to deal with

totality and partiality. A generated profile is marked as total if the corresponding right-hand side is syntactically well-defined. For TGI definitions, profiles in $\mathcal{P}_0$ may artificially be regarded as total. In the complete signature no total profile with $\oslash$ as codomain will remain. (When computing profiles with non-basic domains by means of union, one need not consider basic domains containing $\oslash$, as noticed earlier, and totality requires all involved profiles to be total.) It is easy to show that the strengthened completion algorithm results in a t-regular and sound signature.

EXAMPLE 9.
For the definition in Example 4 of partial subtraction on *Nat* with subtypes *Zero* and *Pos*, the smallest representation of the complete set of profiles is now:

$$
\begin{array}{lll}
\oslash \times Nat \longrightarrow & \oslash & [partial] \\
Nat \times \oslash \longrightarrow & \oslash & [partial] \\
Zero \times Pos \longrightarrow & \oslash & [partial] \\
Zero \times Zero \longrightarrow & Zero & [total] \\
Pos \times Zero \longrightarrow & Pos & [total] \\
Nat \times Zero \longrightarrow & Nat & [total] \\
Zero \times Nat \longrightarrow & Zero & [partial] \\
Nat \times Nat \longrightarrow & Nat & [partial]
\end{array}
$$

3.5 Optimal typing

Optimal typing is concerned with semantic language properties. We define the following versions of optimality:

- The *optimal type* of a term t is the smallest type that would be semantically correct for t. It follows that there is a well-defined ground instance of t of type U for every basic subtype U of its optimal type. In particular, variables are optimally typed. The empty type is optimal for an ill-defined term.
- An *optimal profile* $f : D \longrightarrow C$ is such that an application $f(x)$ has the optimal type C if x is a variable (list without repetitions) of type D. Thus, generator profiles are optimal by definition.
- An *optimal profile set* for a function f with domain D consists of all nonredundant optimal profiles for f with domains included in D.
- An *optimal signature* $\mathcal{S}$ for a function set $\mathcal{F}$ consists of optimal profile sets for the functions in $\mathcal{F}$.

A complete signature is not necessarily optimal, as the following examples show. Let $f, g : Nat \longrightarrow Nat$ be defined as $f(x) == x \dot{-} x$ and $g(x) == \mathbf{PS}x$. The completion algorithm would not be able to discover any of the profiles $f : Nat \longrightarrow Zero$, $g : Pos \longrightarrow Pos$, or $g : Zero \longrightarrow Zero$.

Theorem 4 below expresses sufficient syntactic conditions for optimality. In order to formulate such conditions we define the syntactic notions of *O-term*, whose minimal type will be optimal, and *N-term*, which will have well-

defined instances (being "non-ill-defined"). The two concepts are defined by mutual recursion on term structure:

- An *O-term* is either
  (1) a term of type $\oslash$, or
  (2) a variable, or
  (3) a N-term of basic type, or
  (4) $t\,\mathbf{as}\,T$ or $t\,\mathbf{qua}\,T$ when t is an O-term, or
  (5) an application of an optimally typed function to arguments, each consisting of only variables, terms of type $\oslash$, **as** , **qua** , and generators, such that no variable occurs twice (not adding occurrences in disjoint **if** or **case** branches), and where generators are applied to arguments of minimal types matching the domains of the selected profiles exactly.
- A *N-term* is either
  (1) a syntactically well-defined term of minimal type other than $\oslash$, except that partial applications within O-terms are allowed, provided that for any two such O-terms with common variables one is a subterm of the other, or
  (2) a term which can be transformed into a N-term by the insertion of coercions on subterms.

Insertion of coercions may cause partial applications to become total; for instance for $x : Pos$, the outermost **P**-application in $\mathbf{PP}x$ is partial, but total in $\mathbf{P}((\mathbf{P}x)\,\mathbf{as}\,Pos)$. In the latter, the coercion is inside an O-term, thus the latter, and thereby $\mathbf{PP}x$ also, are N-terms.

Since there are finitely many ways of inserting coercions to avoid a partial application, this can be explored syntactically; in most cases a simple bottom-up analysis, together with repeated type-analysis, will do. (Top-down strategy is not appropriate since an O-term may no longer be an O-term after coercion of internal O-terms.) For instance, the term

$$(x \dot{-} y) \dot{-} (\mathbf{P}x + y)$$

with $x, y : Nat$, becomes syntactically well-defined by first restricting y to $Zero$, making the leftmost subtraction a total application, and x to Pos, making the **P**-application total, and then reducing the type of $\mathbf{P}x$ to $Zero$, making the outermost subtraction-application total. Examples of O-terms are given below.

THEOREM 3. *The minimal type of an O-term is optimal, and N-terms have well-defined instances.*

PROOF: By induction on syntactic structure on terms. For optimality of O-terms, we consider the kinds of O-terms. Cases 1, 2 and 3 are trivial. Case 4: If a subterm t of type T is an O-term, then the term $t\,\mathbf{as}\,T'$ has minimal type $T \sqcap T'$, which is smaller than that of t, and must be semantically optimal, since

T is. The same argument goes for **qua**, under the assumption of the **qua**-proof obligation. Case 5: Consider the application $f(d)$, and assume the profile $f : D \longrightarrow C$ is selected. Let C' be a basic subtype of C. Since f is optimally typed, there must be a value v in D such that f applied to it gives a value in C'. If f is not an **if** construct, the restricted form of d ensures that the set of values of d under all (type-correct) valuations is all D-values. If f is an **if** construct, the most difficult case is when the test is of minimal type *Bool*; the test must then be a *Bool*-variable, possibly with trivial applications of **as** and **qua**. Since this variable may not occur in the branches, the possible values of the **if** construct must contain all D-values. (A similar discussion applies to **case** constructs.)

For proving that N-terms have well-defined instances, we first observe that an O-term coerced to a non-empty resulting type, must have a value for some valuation (by optimality). The same valuation may be used for multiple occurrences of O-terms coerced in the same way and for O-terms without common variables. Thus a term with only total applications, except inside non-strict arguments and such O-terms (including the coercions), will have a value for the same valuation.

Let x, y : *Nat* and z : *Pos*. Some examples of O-terms, as well as non-O-terms, classified according to optimal type, are as follows:

$\oslash$**-terms:** $\mathbf{P}0$, z **as** *Zero*, $\bot + x$ and $0 \dot{-} \mathbf{S}x$ are O-terms,
but not x **as** *Zero* $+ (x$ **as** *Pos*$)$ and $\mathbf{PPS}0$.

True-**terms:** $0 \leq x$, $0 \leq (x \dot{-} \mathbf{S}x)$ and x **as** *Zero* $\leq x$ **as** *Pos* are O-terms,
but not $x = x$ and $x \leq x$.

False-**terms:** x **as** *Zero* $= \mathbf{S}x$ and $\mathbf{S}x \leq 0$ are O-terms,
but not x **as** *Pos* $\leq x$ **as** *Zero* and $\mathbf{S}x \leq x$.

Zero-**terms:** $(x \dot{-} y)$ **as** *Zero*, x **as** *Zero* $* y$, $\bot * 0$, $0 \dot{-} x$ are O-terms,
but not $x \dot{-} x$ and $\mathbf{PPSS}x$.

Pos-**terms:** $z + z$, $x + (y + z)$, $\mathbf{S}x + \mathbf{P}y$, $\mathbf{S}(x + \mathbf{P}y)$, $\mathbf{S}(x \dot{-} x)$, $\mathbf{S}((x + y) \dot{-} y)$
are O-terms, but not $\mathbf{PSS}x$.

Nat-**terms:** $x + y$, $x \dot{-} \mathbf{S}y$, $0 \dot{-} x$, x **as** *Pos* $\dot{-} y$, $\mathbf{PS}x$ and $\mathbf{PP}x$ are O-terms,
but not $x + x$ and $\mathbf{PPP}x$.

More O-terms of type *Pos* can be obtained by applying **S** (any number of times) to the above *Pos*-, *Zero*- and *Nat*-terms, as well as to $\mathbf{PS}x$, $x + x$ and $x \dot{-} x$, none of which are O-terms.

Generator functions and equality on a type with one-one generator basis as well as on a formal type can be considered optimally typed, provided profiles expressing strictness properties are included. And the **if** construct may be regarded as a function with an optimal profile set, consisting of

$\oslash \times A \times B \to \oslash$
$True \times A \times B \to A$
$False \times A \times B \to B$
$Bool \times A \times B \to A \sqcup B$

for related types A and B. For instance, the term $\mathbf{if}\, x \leq y\, \mathbf{th}\, x + z\, \mathbf{el}\, y + z\, \mathbf{fi}$ of type Pos is optimally typed for $x, y : Nat, z : Pos$, using the optimal profile $\hat{}+\hat{}: Nat \times Pos \longrightarrow Pos$.

THEOREM 4. *A complete signature (or profile set) is optimal, provided the right-hand sides of the defined functions (considering the* **case***-free version of the function definitions), are O-terms for all basic type assignments to the outermost arguments in the left-hand side, when the functions under definition are taken as optimally typed.*

PROOF: One may prove by induction on the number of iteration steps that the codomain in each profile is smaller than or equal to the optimal one. The base case is trivial. To prove the induction step, we use the fact that reducing the type of a subterm cannot make the overall type larger. Since it has been established earlier that a complete signature is sound, the theorem follows.

It is possible to strengthen the theorem somewhat by modifying the fixpoint algorithm to recognize optimal applications of functions.

NOTE: For all the TGI defined functions occurring in the examples of this paper, signature completion results in complete profile sets, even with basic types introduced for each generator profile! Furthermore, they satisfy the syntactic criteria of Theorem 4, except *sum* of Example 8 (due to the lack of the splitting property) and some Boolean functions in Section 3.2 (due to non-trivial use of $\wedge$, $\vee$, $\Rightarrow$).

Another approach to signature completion is given in [2] taking the least fixed point of a rewrite system transformation, using a notion of abstract domains. Thus, this method is based on rewriting rather than type analysis. Soundness, but not optimality results, are proved.

4. Semantic subtypes

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

⟨subtype definition⟩ ::= **type** ⟨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 include 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 (schema) and $U_1,\ldots,U_m$, if any, are type expressions in the formal parameters and defined types. M is the subtype module. Any instance $\bar{T}$ of the left-hand side is by definition a semantic subtype of the corresponding instance $\bar{U}$, such that $\mathcal{V}_{\bar{T}} = \{x \in \mathcal{V}_{\bar{U}} \mid R(x)\}$, where $R \in \mathcal{F}_{\bar{U}}$. $\mathcal{F}_{\bar{T}}$ inherits the functions in $\mathcal{F}_{\bar{U}}$ and contains in addition those introduced in the corresponding instance of the module 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 M, is implicitly declared. 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 declared 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 $V \prec U$.

The **in**-test for a semantic subtype is an application of the restricting predicate.

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 10.
We define a semantic subtype of *Nat* as defined in Example 5, bounded by an unspecified number n (a meta-variable).

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 $\hat{}+\hat{}$, $\hat{}\dot{-}\hat{}$: *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 7.) 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 $\hat{}\dot{-}\hat{}$ 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.

The soundness results from the section on syntactic subtypes, i.e. semantical correctness of a complete signature and semantical well-definedness of a syntactically well-defined term, generalize to the case where both kinds of subtypes are present. (This is easily seen by inspection of the proofs.) Obviously, the "splitting property" must be satisfied in order to speed up the signature completion as explained above.

Even though it is obvious that membership in semantic subtypes cannot in general be determined syntactically, the optimality results stated in Theorem 3 and 4 do hold in the case where both kinds of subtypes are present, provided convexity, disjointness, and emptiness of subtypes are syntactically known (possibly with the help of lemmata proved by the user), and provided the occurrence of "basic type" in the definition of O-term is replaced by "non-empty minimal type". (It is assumed that terms of empty types are given minimal type $\oslash$.) Additional semantic subtypes must be added so that each non-empty type can be seen as the disjoint union of minimal subtypes.

For instance, the addition of the subtype *BNat* above does not destroy optimality of the (re)completed profile sets for the *Nat*-functions $\hat{}+\hat{}$, $\hat{}\dot{-}\hat{}$ and $\hat{}\leq\hat{}$; but some new profiles will be generated, including $\hat{}\dot{-}\hat{} : BNat \times BNat \longrightarrow BNat$ (for the version of $\hat{}\dot{-}\hat{}$ belonging to $\mathcal{F}_{Nat}$). And terms such as $\mathbf{SS}x$ and $x{+}\mathbf{S}y$ remain O-terms for $x, y : Nat$ and are thus optimally typed *Pos*-terms (by point 5 of the definition of O-term), but $\mathbf{SS}0$ and $\mathbf{SSS}0$ are no longer O-terms (because point 3 of the definition of O-term no longer applies since *Pos* now has a smaller non-empty subtype).

4.1 Canonical generator terms

We show 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.

Consider the concept $Set\{T\}$, as defined in Example 3. A one-to-one property may be 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 $\hat{}< \hat{}: T \times T \longrightarrow Bool$. (The formalization of such assumptions is beyond the scope of the present 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, see also [13].

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 $y < x$ **th** $add(s, x)$ **at** Set **el if** $y = x$ **th** s
    el $add(add(s', x), y)$ **at** Set **fi fi fo qua** $CSet$
  one-one
  - - - - - - - -
endmodule

The discriminators of a **case** construct by definition refer to generators, not to possibly 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 of *CSet* asserts a one-to-one relation between *CSet* generator terms and abstract sets. Equality is redefined accordingly:

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

The required verification of this redefinition consists in proving:

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

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

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

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

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.

Another approach to define a one-to-one generator basis for data types such as sets, is to use semantic subtypes to restrict the domain of generators, for instance by restricting the domain of the *add* generator to be

$$\{(s,x){:}\ Set{\times}T \mid \textbf{case}\ s\ \textbf{of}\ \emptyset \to true \mid add(s,y) \to y{<}x\ \textbf{fo}\}$$

where $<$ is a total order on T. This approach has been suggested in [15] in an extension to the Vienna Development Method; and may be adopted in other languages supporting subtypes of this kind, including ABEL. It gives a concise definition; however, the *add* generator is partial and does not capture the traditional add operator on sets; the latter would have to be added as a defined operator, and its totality would require a proof. In addition it would be desirable to prove that the basic *add* is an approximation of the total one, as this would allow for standard case definitions of other set functions.

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

4.2 Definedness Control in Term Rewriting

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 $\perp \to \perp$.

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 5. *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 the 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 11.
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** $\perp \to true \mid 0 \to true \mid \mathbf{S}x' \to$

case x' **of** $\bot \to \mathit{false} \mid$ **others** $\to \mathit{canonic}(x')$ **fo fo**

(Notice that the $\bot$ of $\mathit{Nat}_\bot$ is regarded as an ordinary generator.)

We define:

type $\mathit{Nat}' == x{:}\,\mathit{Nat}_\bot$ **where** $\mathit{canonic}(x)$ **convex**

Then the user accessible successor function is the following redefinition of the $\mathit{Nat}_\bot$ generator $\mathbf{S}$, renamed for perspicuity:

func $\mathbf{S}'{:}\ \mathit{Nat}' \longrightarrow \mathit{Nat}'$

def $\mathbf{S}'x ==$ **case** x **of** $\bot \to \bot \mid$ **others** $\to \mathbf{S}x$ **fo qua** Nat'

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

Consider the operator $\hat{}\leq\hat{}$ as defined in Example 7. The following extended set of rewrite rules will be generated for Nat':

R1: $0 \leq y == \mathit{true}$
R2: $\mathbf{S}x \leq 0 == \mathit{false}$
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 now 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 syntactically well-defined expression. 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 [17] 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 with respect to certain left-hand arguments. In fact, that is the case with the left argument of 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. Then, $\mathbf{S}'f(x) \leq \mathbf{S}'0$ can be reduced to $f(x) \leq 0$ even for partial f, using R7 and the new rule.

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. A mathematically simple and controllable treatment of subtypes and partial functions is lacking in Larch, and this causes problems [14].

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 associated proof obligations, if any, are expressed in terms of explicit first order formulas.

The PVS language is also designed to be constructive in part, [22, 23]. 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.

In the PVS approach of strong typing user defined functions are assumed to be strict and the boolean operators to be left-strict. The kind of non-strictness allowed in ABEL 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, [24]. 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 $0 \leq y == \mathit{true}$ from Example 11 would require y to be well-defined (when generated by stratification). Thus, the ABEL notion of weakly correct rewriting (respecting $\sqsubseteq$) provides useful reasoning about partly well-defined terms, which are irreducible in OBJ.

Acknowledgements

The authors are indebted to many colleagues for fruitful discussions and feedback, and in particular to Bjørn Kristoffersen and Anne Salvesen. In addition Friedrich von Henke has given useful feedback. We are grateful for many valuable comments by the referees.

References

[1] T.J. Bastiansen: "Parametric Subtypes in ABEL." Research report 207, Dept. of Informatics, University of Oslo, 1995.
[2] D. Bert, R. Echahed: "Abstraction of conditional term rewriting systems." Research Report 942, LGI-IMAG, Grenoble, France, 1995.
[3] O.-J. Dahl: Can Program Proving be Made Practical? In *Les Fondements de la Programmation*, M. Amirchahy and D. Néel, (Eds.), INRIA, 1977
[4] O.-J. Dahl: *Verifiable Programming.* Prentice Hall 1992.
[5] 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.
[6] 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.
[7] O.-J. Dahl, B. Myhrhaug, K. Nygaard: "Simula 67 Common Base Language", NCC pub. S-22, Norwegian Computing Center, 1971.
[8] M. Elvang-Gøransson, O. Owe: "A simple sequent calculus for partial functions." *Theoretical Computer Science* 114:317-330, 1993.
[9] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, J. Meseguer: "Principles of OBJ2." In *Proceedings, 1985 Symposium on Principles of Programming Languages and Programming*, W. Brauer, Ed., LNCS 194, Springer-Verlag, 1985, pp. 52-66.
[10] J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, J.-P. Jouannaud: "Introducing OBJ", Oxford University Computing Laboratory, 1993.
[11] J.V. Guttag: "The Specification and Application to Programming of Abstract Data Types." Ph. D. Thesis, Computer Science Department, University of Toronto, 1975.
[12] J.V. Guttag, J.J. Horning, J.M. Wing: "Larch in Five Easy Pieces." Digital Systems Research Center, Palo Alto, CA, July 1985.
[13] J.E. Hannay: "Hiding Symbols in Equational Reasoning." Unpublished, available by `http://www.ifi.uio.no/~joh/lhide.ps`, 1995.
[14] J.J. Horning: "The Larch Shared Language: Some Open Problems." In M. Haveraaen, O. Owe, O.-J. Dahl (Eds.): *Recent Trends in Data Type Specification.* LNCS 1130, Springer Verlag, 1996, pp. 58-73.
[15] C.B. Jones, C.A. Middelburg: "A Typed Logic of Partial Functions Reconstructed Classically." *Acta Informatica* 31: 399-430, 1994.
[16] B. Kristoffersen, O.-J. Dahl: On Introducing Higher Order Functions in ABEL. *Nordic Journal of Computing*, 5(1998).
[17] O. Lysne, O. Owe: "Definedness and Strictness in Generator Inductive Definitions." Part of Lysne's dr. scient. thesis, Univ. of Oslo, Dec. 1991.
[18] R. Milner: "A Proposal for Standard ML." Report CSR-157-83, Computer Science Dept., Edinburgh University, 1983.
[19] R. Nakajima, T. Yuasa (Eds.): *The IOTA Programming System.* LNCS 160, Springer Verlag, 1983.
[20] O. Owe, O.-J. Dahl: "Generator Induction in Order Sorted Algebras." *Formal Aspects of Computing*, 3:2-20, 1991.
[21] O. Owe: "Partial Logics Reconsidered: A Conservative Approach" *Formal Aspects of Computing*, vol. 5, 1993, pp. 208-223.

[22] S. Owre, N. Shankar, J.M. Rushby: "The PVS Specification Language." Computer Science Lab., SRI International, Menlo Park, CA, 1993.
[23] N. Shankar, S. Owre, J.M. Rushby: "A Tutorial on Specification and Verification Using PVS." Computer Science Lab., SRI International, Menlo Park, CA, 1993.
[24] 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.
[25] D.A. Turner: "Elementary stong functional programming." In *Functional Programming Languages in Education*, P.H. Hartel, R. Plasmeijer (Eds.), LNCS 1022, Springer 1995, pp. 1-14.