Department of informatics

On Introducing Higher Order Functions in ABEL

Ole-Johan Dahl
and Bjørn
Kristoffersen

Research Report 210

ISBN 82-7368-123-8
ISSN 0806-3036

December 22, 1995

Abstract

We discuss how the 1'st order specification and programming language ABEL could be extended with higher order functions. Several issues arise, related to subtyping, parameterization, strictness of generators and defined functions, and to the choice between lambda expressions and currying. The paper can be regarded as an exercise in language design: how to introduce higher order functions under the restrictions enforced by (1'st order) ABEL. A technical result is a soundness proof for covariant subtype replacement, useful when implementing data types under volume constraints imposed by computer hardware.

Contents

1 Introduction

The specification and programming language ABEL (Abstraction Building Experimental Language) has been under development at the University of Oslo since the late 70's [DLO86, Owe91, Dah92]. The applicative kernel of ABEL is a typed 1'st order language with subtypes and partial functions. Specifications may consist of typed 1'st order axioms (with loose semantics); however, we are here concerned with the so called TGI fragment of ABEL, in which types and functions are defined inductively over given sets of generator functions. TGI stands for *terminating generator induction.* Partial functions may be defined using an explicit error indicator.

A recent paper [DO95] is a prerequisite for a thorough understanding of what is to follow. It explains the notion of TGI, as well as some applications of subtypes. The TGI fragment of ABEL can be seen as a ML-like, 1'st order programming language. With this paper we attempt to extend the TGI fragment with higher order functions.

By a *functional language* we shall mean an applicative language where functions are "values"; and may as such be bound to variables, passed as parameters and returned as results from (other) functions. Then function profiles can be taken as bona fide types, although the associated value sets are not in general definable inductively. It makes sense to have functional *imperative* languages, and even higher order procedures, but this is outside the scope of the paper.

The most important motivation for *programming* with higher order functions is reuse of code. Certain functionals, such as *map*, *filter* and *fold*, are to the functional programmer what control structures are in imperative languages. Since those functionals can be defined within the language, it is easy to introduce new "control structures", e.g. *map*2 for distributing a binary function over two lists instead of one. The usefulness of higher order functions is further increased when combined with some kind of "polymorphism". Whether higher order constructs are useful also in formal specification and mechanical theorem proving, is more uncertain. Some restrictions on the use of higher order functions will be described. Our goal is to lift the methodology of terminating generator induction in ABEL to a higher order setting, and the following can be read as an attempt to identify and motivate a certain *style* of higher order specification.

2 Higher order TGI

Currying is one way to introduce higher order functions. Extending pure combinatory logic [HS86] with currying and pattern matching gives a relatively simple model of functional languages, which can be implemented by term rewriting. In contrast to lambda calculus based approaches [Bar84, Bar92], combinators do not need the complexity related to substitution and local variables. Higher order generator inductive specifications are then obtained as follows:

- A type constructor $\rightarrow$ is introduced for writing functional types.

Since the codomain of a function can now be a function space, currying follows. Whereas 1'st order types are specified through a set of generator functions, functions are not (in general) finitely generated. Thus, there is no induction principle for functional types.

The TGI style of function definition requires recursion to be "guarded" by induction on some argument. Thus, for higher order functions recursive definitions will require at least one 1'st order argument. The following is an extract from a higher order TGI specification:

$\quad$ **func** $map : \ (Nat \rightarrow Nat) \rightarrow Seq\{Nat\} \rightarrow Seq\{Nat\}$
$\quad$ **def** $map\ F\ q\ \ ==$ **case** q **of** $\varepsilon \Rightarrow \varepsilon \mid q' \vdash x \Rightarrow (map\ F\ q') \vdash (F\ x)$ **fo**
$\quad$ **func** $incr : \ Seq\{Nat\} \rightarrow Seq\{Nat\}$
$\quad$ **def** $incr == \ map\ (add\ 1)$
where
$\quad$ **func** $add : \ Nat \rightarrow Nat \rightarrow Nat$
and sequences have the generators ε (empty sequence) and $\hat{}\vdash\hat{}$ (append right).

Function applications are written by juxtaposition, $e_1\, e_2$. Thus, the traditional 1'st order notation, $f(e_1, \ldots, e_n)$, is a special case when the argument is a tuple (i.e. the domain is a Cartesian product). As usual, application is left associative, whereas the type constructor $\rightarrow$ is right associative. Every function, be it a declared function or a function expression, takes a single argument (possibly a tuple); however, if the function value is again a function, another argument may be added, and so on recursively.

Definition 2.1

1. The *depth* of a non-functional type is equal to 0. The depth of a functional type is one greater than that of its codomain part.
2. The *c-arity* of a function is equal to the depth of its profile.
3. An application of type T is said to be *complete* if T is non-functional.
4. The type of a complete application of a function is called the *c-codomain* of the function.

Thus, the c-arity of *map* is equal to 2, its c-codomain is $Seq\{Nat\}$ and the left hand side of its definition is a complete application.

The definition of a function f has the following general format:

def $f\ x_1 \ldots x_n == e$

The left hand side has the form of an application of f, possibly incomplete, where the arguments are distinct variables or variable tuples, implicitly typed as required by the type ("profile") of f, and the right hand side is an expression in $x_1, \ldots, x_n$. We require that recursive applications in the right hand side are "guarded" by generator induction on the argument(s) in question.

In a language allowing lambda expressions, the above could be taken as a shorthand for:

let rec $f\ == \ \lambda x_1 . \ldots \lambda x_n . e$

The function values we allow are those obtained from incomplete applications of *named* functions. This corresponds to restricting lambda expressions in right hand sides to the outermost level. Remaining occurrences of lambda expressions in typical functional specifications tend to be rather trivial, e.g. functions for swapping the components of a pair in order to compose two given functions. We have found that specifications become more readable by insisting that such operations are written as (incomplete) applications of named combinators. Observe also that naming is a necessary condition for the reuse of code.

2.1 Term rewriting

The lambda restriction has the advantage that specifications can be assigned a semantics by translation into so called *curryfied term rewriting systems* (CTRS), as defined by van Bakel and Fernández [vBF93], without first doing lambda lifting [Joh85].

As a first step in the translation from an ABEL specification to a CTRS, function definitions are augmented with extra parameters so that left hand sides are complete applications. Case expressions in the right hand side are then removed by writing a function definition as a set of equations, e.g. for *map* above (disregarding the convention that **case** constructs are strict in the discriminand, see [DO95]):

E1: $map\ F\ \varepsilon\ ==\ \varepsilon$
E2: $map\ F\ (q' \vdash x)\ ==\ (map\ F\ q') \vdash (F\ x)$

A *pattern* p is a linear expression of non-functional type built from generators and variables. We insist that non-variable patterns are on the form $g\ p_1 \dots p_n$. In particular, a function variable cannot be the leftmost-innermost term of a compound pattern. Left hand sides of equations are easily seen to be built from patterns and a single non-generator function symbol.

The next step is to translate such equations into CTRS rules (oriented from left to right). For this we need to replace applications of function *expressions* by applications on the form $(f^n\ e_1 \dots e_n)$, for f^n a function symbol of arity n.

Every CTRS has a special binary function symbol ap^2 (pronounced "apply"). Furthermore, for f a function symbol of c-arity n, $n+1$ function symbols $f^0, \dots, f^n$ are introduced in the corresponding CTRS. The relation between a set of function symbols $f^0, \dots, f^n$ are defined by a set of *curry rules*, one for every $i < n$:

$$(ap^2\ (f^i\ x_1 \dots x_i)\ x_{i+1}) \quad \rightarrow \quad (f^{i+1}\ x_1 \dots x_i\ x_{i+1})$$

The translation $\overline{e}$ of an expression e is defined inductively, where $e\!\downarrow$ denotes the normal form of e with respect to the curry rules:

$$\begin{aligned} \overline{x} &= x \\ \overline{f} &= f^0 \\ (\overline{e_1\ e_2}) &= (ap^2\ \overline{e_1}\ \overline{e_2})\downarrow \end{aligned}$$

Applying the expression translation to the ABEL equations gives a set of *proper* rewrite rules. Observe that we may need to use ap^2 with a number of distinct types, all on the general form $(S \rightarrow T) \rightarrow S \rightarrow T$. However, given that all type dependent function overloading has been resolved prior to the translation to CTRS, the overloading of ap^2 is irrelevant.

Lemma 2.2 Let R be a CTRS obtained from an ABEL specification as described above. Then there are no occurrences of ap^2 in the left hand side of proper rules in R.

Proof: Consider a function definition, where the right hand side e does not contain case expressions:

def $f\ x_1 \dots x_n == e$

Since the left hand side is complete, the c-arity of f is n. Hence the result of translating the left hand side is $(ap^2 \dots (ap^2\ f^0\ x_1) \dots x_n)\downarrow = (f^n\ x_1\ \dots x_n)$.

If the right hand side contains a case expression, the left hand sides of the corresponding ABEL equations will contain patterns. But since non-variable patterns are always complete applications, the translation of a compound pattern is the following:

$$\begin{aligned} \overline{(\dots(g\ p_1)\dots p_n)} &= (ap^2 \dots (ap^2\ g^0\ \overline{p_1})\ \overline{p_n}) \\ &= (g^n\ \overline{p_1} \dots \overline{p_n}) \end{aligned}$$

Arguing by induction on expressions, each $\overline{p_i}$ for $1 \leq i \leq n$ can be assumed to be free for occurrences of ap^2. □

The proper rules corresponding to a given function f are from the above seen to be associated with f^n, where n is the c-arity of f. For *map* we obtain:

R1: $map^2\ F\ \varepsilon\ \rightarrow \varepsilon$
R2: $map^2\ F\ (q' \vdash x) \rightarrow (map^2\ F\ q') \vdash (F\ x)$

Remark 2.3 Given a suitable notion of *matching*, ABEL equations could be oriented and used directly as rewrite rules, without going through the final translation into a CTRS. This would give a deeper term structure, and probably result in a loss of efficiency. We further find that CTRSs provide a simple explanation of currying. Observe also that the representation of ABEL definitions as CTRS rules can be easily hidden from the user by a "pretty-printer" facility.

2.2 Termination

Generators with functional domains pose a problem to syntactic termination checks. Consider the following (not very meaningful) specification extract:

func $g1 : T$
func $g2 : (T \rightarrow T) \rightarrow T$
genbas $g1, g2$
func $f : T \rightarrow T$
def $f\ x\ ==$ **case** x **of** $g1\ \Rightarrow\ g1 \mid g2\ F\ \Rightarrow\ F\ (g2\ F)$ **fo**

There is no recursive applications in the definition of f, still we have the following infinite computation (in the corresponding CTRS):

$$f^1\ (g2^1\ f^0)\ \Rightarrow\ ap^2\ f^0\ (g2^1\ f^0)\ \Rightarrow\ f^1\ (g2^1\ f^0)\ \Rightarrow\ \ldots$$

If generators are not allowed to take functional parameters, and the depth of every recursive application (not occurring as operator in a function application) is equal or greater than the depth of the corresponding left hand side, we conjecture that currying does not have any effect on termination. If that is correct, 1'st order termination checks can be lifted to higher order. Termination of higher order TGI specifications is studied in a forthcoming paper [Kria].

2.3 Strictness control

In 1'st order ABEL, the empty type $\oslash$ is considered to be a subtype of every type. The special symbol $\perp$ is an expression of type $\oslash$, thus legal in any argument position. It has no value and is "ill-defined" in that sense. Within the TGI fragment of ABEL $\perp$ can be seen as a simulation of an infinite computation.

ABEL adopts a partly non-strict semantics for *defined* functions whereas *generators* are strict. Non-strictness is motivated by the use of term rewriting. For instance, the definition:

func $fst : T \times U \rightarrow T$
def $fst(x, y) == x$

would allow any application $\mathit{fst}(e_1, e_2)$ to be rewritten to e_1, even if e_2 is an ill-defined expression. On the other hand, expressions in generators are considered evaluation results and, as such, cannot have ill-defined components. Generator strictness can be naively implemented by adding rules on the form $(g\ \perp) \rightarrow \perp$. Adding currying to this mixed strictness strategy in a clean way is a challenge.

As a first step we let $\oslash$ remain a subtype of all types, including functional ones. Let $f : S \rightarrow T$ be a function ill-defined on the whole of S. Then the strong (i.e. non-strict) equality $\perp\ x == f\ x$ holds for every $x : S$. In spite of this, f and $\perp$ should not be considered equal; the former is a value, whereas the latter is not. Although any application of f is ill-defined, the function f is not itself ill-defined.

Remark 2.4 Observe that disallowing $\perp$ to be used with function types would complicate the type structure. ABEL allows for the specification of parameterized types, e.g. $\mathit{Seq}\{T\}$ denotes sequences of type T, where T is a *formal type.* If $\perp$ was used in a type T position in such a setting, T could not be instantiated with function types. Hence, we would be forced to introduce two kinds of formal types: those ranging over all types and those ranging only over non-functional types.

As already mentioned, the standard notation for higher order function application makes it necessary to look at an ABEL notation such as $f(x, y)$ as an application of f to a tuple. Furthermore, a tuple is an application of the generator of a Cartesian product type. Since generators are strict, we have to revise the notion that e.g. the function *fst* above can be strict in x and non-strict in y. This does lead to a certain overhead for strongly correct term rewriting, cf. [DO95], however, we may retain the notion of mixed strictness *for curried functions.* For instance, the function:

func $\mathit{fst}_c : T \rightarrow U \rightarrow T$
def $\mathit{fst}_c\, x\, y == x$

should be strict in x, but non-strict in y. Thereby the Curry isomorphism is violated:

$$T \times U \rightarrow V \not\equiv T \rightarrow U \rightarrow V$$

But, on the other hand, we are able in this way to exploit a notational difference for a useful semantic purpose.

3 Parameterization and higher order functions

"Functions as first-class citizens" might serve as an useful guide line to the design of programming languages. For formal specification languages, however, one could argue against this strive for "equal rights for all values" on several grounds. First and foremost, formal reasoning does not always generalize to higher types. J. Gougen [Gou90] refers to the undecidability of higher order unification as a prime example.

The fact that functions are not in general finitely generated, implies that certain important functions, such as the *equality relation* are not generally computable over function spaces. 1'st order ABEL specifies that equality exists for all types, including formal ones. For that reason we insist that higher order equality should be part of the extended language, although not of the constructive fragment. See also chapter 6.

Unrestricted rights for function values also make subtyping rules for parameterized types more complex. We return to subtyping in chapter 4.

3.1 "Formal" functions

In a certain sense it is possible to express higher order specifications in 1'st order ABEL. Using loose axiomatic function specifications in a module parameterized by types, implicit functional arguments can be simulated. Consider the sorting of sequences of an unspecified element type T. This type must be ordered by some binary predicate $\hat{}\leq\hat{}$, as expressed by the following ABEL module:

```
**property** $SortOrd\{T\}$ $==$
**module**
    **func** $\hat{}\leq\hat{} : T \times T \to Bool$
    **axm** $x, y, z : T \bullet$
        $x \leq y \vee y \leq x$
        $x \leq y \leq z \Rightarrow x \leq z$
**endmodule**
```

The axioms do not define a unique function. The purpose of the *SortOrd* module is rather to state minimal requirements on some formal type in a parameterized module, such as the following sorting module:

```
**funcs** $SortSeq\{T\}$ **using** $SortOrd\{T\}$ $==$
**module**
    **func** $insert : Seq\{T\} \times T \to Seq\{T\}$
    **def** $insert(q, x)$ $==$ **case** $q$ **of** $\varepsilon \Rightarrow \varepsilon \vdash x \mid q' \vdash y \Rightarrow$
        **if** $x \leq y$ **then** $insert(q', x) \vdash y$ **else** $q \vdash x$ **fi** **fo**
    **func** $sort : Seq\{T\} \to Seq\{T\}$
    **def** $sort(q)$ $==$ **case** $q$ **of** $\varepsilon \Rightarrow \varepsilon \mid q' \vdash x \Rightarrow insert(sort(q'), x)$ **fo**
**endmodule**
```

The *SortSeq* module can be instantiated by replacing the formal type T by some actual type, say *Int*. The actual type must fulfill the requirements on the formal type T; i.e. there must exist a binary predicate satisfying the axioms given in module *SortOrd*:

```
**funcs** M
    **using** $Int$, $SortSeq\{Int\}$
**module** ... **endmodule**
```

The actualization of $\hat{}\leq\hat{}$ is a kind of static parameter transmission of a functional value at the module level. ABEL allows for the renaming of loose functions, which makes them more parameter-like. For that reason they are sometimes called *formal* functions.

The operational aspect of the above specification could in a higher order language be encoded by making the formal function $\hat{}\leq\hat{}$ an argument (in the usual sense) to the *sort* function:

```
**func** $sort : (T \times T \to Bool) \to Seq\{T\} \to Seq\{T\}$
**def** $sort$ $\hat{}\leq\hat{}$ $q$ $==$ ...
```

The following should be observed:

1. The requirements on the ordering relation are lost in the higher order encoding of the *sort* function.
2. A formal function as above can be an implicit parameter to several functions at the same time. One could for instance include alternative sorting algorithms in the *SortSeq* module. They would all be instantiated by one module expression.

So, the higher order extension of ABEL does not remove the need for formal functions. On the other hand, in many cases we may not wish to restrict a formal function more than what is required by the profile. To simulate the functional argument to *map* using a formal function requires much notational overhead including explicit module instantiation and function renaming. J. Gougen [Gou90] argues strongly for the simulation of higher order functions by parameterized modules, and proposes techniques to reduce the notational overhead involved in actualization of formal functions (although he does not use our terminology). To some extent he succeeds, but only at the expense of a rather complicated set of rules for function renaming and so called "default views". Furthermore, such simulation is only possible to the extent that the set of actual higher order arguments is decidable statically.

Although formal reasoning does not always generalize to higher order, the practical significance of this varies from one specification language to another. For example, equational reasoning in the TGI fragment of ABEL is not based on completion procedures. Hence, the undecidability of higher order unification need not be a problem. Moreover, unification is known to be decidable for the class of *higher order patterns* [Mil91]. The left hand sides of TGI rewrite rules lie within this class.

Higher order functions invite users to build specifications by "putting functions together". In a 1'st order language, every functional composition needs a name, which leads to another style of specifications. According to Gougen parameterization is useful for specification "in the large", whereas higher order functions are used as a structuring mechanism "in the small". We agree, but maintain that both are needed. Further motivation for "why functional programming matters" can be found in Hughes [Hug90].

3.2 Polymorphism versus parameterized modules

Higher order functions are particularly useful if allowed to be used with different types, such as in ML style polymorphism. We feel, however, that explicit function typing is generally an important part of system design and specification documentation. Furthermore ML style type inference does not go well together with subtyping. It turns out that parameterized modules offer the same expressiveness in a way better adapted to ABEL style typing.

The most frequent use of higher order functions is to "lift" operations from some type T to a compound data structure containing T values. The *map* functional is a prime example. Assume access to a parameterized module defining *map*, as in chapter 2, and consider a given application $(map\ f)$. From the type of f, the type checker will know how to instantiate the module. Thus, there is no notational overhead caused by explicit module instantiation involved in obtaining polymorphism:

- Instantiations of parameterized modules follow from the types of corresponding function applications, and can hence be implicit. There is no need for additional syntactic sugar.
- Parameterized modules are as expressive as polymorphism. For every function f defined in a polymorphic program P, the number of distinct types that are assigned to occurrences of f in computations of P is finite and can be computed statically.

4 Subtyping in higher types

For every (1'st order) type defined inductively over a one-to-one generator basis, the value set can be divided into disjoint subsets by introducing a list of *basic* subtypes, and using those subtypes as codomains of the generators. For types with a many-to-one generator basis, a proof of disjointness is required. *Intermediate* subtypes arise from taking unions of basic types. This hierarchy, or type family, becomes a complete lattice by adding the empty type $\oslash$ as minimal element. Such *syntactic subtypes* are motivated and described in [DO95].

ABEL also supports *semantic subtypes*, for which the value set of a given type may be restricted by some predicate and the set of associated functions may be extended and partly redefined, typically:

$$\begin{array}{ll}\textbf{type}\ T\{\ldots\} == & x : U\ \textbf{where}\ P \\ & \textbf{module}\ \langle\text{function definitions/redefinitions}\rangle\ \textbf{endmodule}\end{array}$$

where U is a 1'st order type expression in defined types and the parameters to T, and P is a predicate in x and functions associated with U. Our task is to extend the subtype relation to higher order. Notice that only 1'st order subtypes can be defined (viewing formal type parameters, if any, as 1'st order types).

4.1 Covariance versus contravariance

The first question to be answered is the following: When is one function type, say $T' \rightarrow U'$, a subtype of another, $T \rightarrow U$? The codomain should clearly be covariant: $U' \preceq U$.

The subtype relation on 1'st order types is designed so that values of a type T may be used with type U for $T \preceq U$, without explicit coercion. We say that 1'st order types are *coercion-free upwards*. To maintain this property in higher order, we need the following *contravariant* subtype rule:

$$\textbf{Contra:}\quad \frac{T \preceq T'\ ,\quad U' \preceq U}{T' \rightarrow U' \preceq T \rightarrow U}$$

The rule is called contravariant since the direction of the subtype relation on function types is reversed compared to the relation on the domains. Contravariance renders typing *sound* in the sense that well-typed expressions in total functions are well-defined.

Lemma 4.1 The contravariant subtyping rule respects coercion-freedom upwards.

Proof: Let $f : T \rightarrow U$ be a formal parameter and $f' : T' \rightarrow U'$ a function. Given $T \preceq T'$ and $U' \preceq U$, f' is a legal actual argument for f. For $t : T$ the expression $f'(t) : U$ is well-typed. It is well-defined because $T \preceq T'$ gives that $f'(t)$ returns a value in U', and $U' \preceq U$. □

The following semantic interpretation ensures that subtypes correspond to subsets on value sets, given a contravariant subtype rule:

Definition 4.2 A function type $T \rightarrow U$ denotes the set of functions f, which given a value v in T, either is ill-defined or returns a value in U. The behaviour of f is undefined for values outside T.

Functions will never be applied outside their domains in well-typed expressions, so such applications need not be assigned any meaning. The term "undefined" should thus be understood as "unspecified" rather than "ill-defined".

The contravariant subtype rule is needed to obtain coercion-freedom upwards. This does not mean that contravariance is always the natural choice; at first it looks counter intuitive for most people. Consider:

```
**type** $Mapping\{X,Y\}$ ==
**module**
   **func** $init :\ Mapping$
   **func** $\hat{}[\hat{} \mapsto \hat{}] :\ Mapping \times X \times Y \rightarrow Mapping$
   **genbas** $init, \hat{}[\hat{} \mapsto \hat{}]$
   **func** $\hat{}[\hat{}] :\ Mapping \times X \rightarrow Y$
   **def** $M[x]$ == **case** $M$ **of** $init \Rightarrow \perp \mid M'[x' \mapsto y] \Rightarrow$
                            **if** $x' = x$ **then** $y$ **else** $M'[x]$ **fi** **fo**
   **obsbas** $\hat{}[\hat{}]$
   **func** $dom :\ Mapping \rightarrow Set\{X\}$
   **def** $dom(M)$ == **case** $M$ **of** $init \Rightarrow \emptyset \mid M'[x \mapsto y] \Rightarrow add(dom(M'), x)$ **fo**
**endmodule**
```

Assume that $M : Mapping\{T,U\}$ and $M' : Mapping\{T',U\}$ for $T \preceq T'$. If *Mapping* were contravariant in its first parameter, it would be legal to substitute M' for M. But then $dom(M) : Set\{T\}$ is unsound.

The encoding of function types by finitely generated mappings is not a true simulation. The "typing anomaly" depends on the possibility of observing the *range* of *Mapping* values. This cannot be done for function types, so *Mapping* is not a counter example to the soundness of contravariance. But the example does demonstrate a mismatch between contravariance and an intuitive notion of subtyping.

When redefining functions in (semantic) subtypes it is natural, or even necessary, to restrict the domains.

Consider the subtype $BNat = \{x : Nat \bullet x \leq Max\}$ of natural numbers bounded from above by some constant Max. To avoid uncontrolled overflow during run-time, we should redefine, say addition to explicitly return $\perp$ when the result is too large. The profile for $\hat{} + \hat{}$ should hence be:

func $\hat{}+\hat{}$: $BNat \times BNat \rightarrow BNat$

According to the principle of contravariance, this redefinition is not a legal actual argument for a formal parameter of type $Nat \times Nat \rightarrow Nat$.

Redefinitions of functions in subtypes result in a special kind of *overloading.* The proper function binding for an application is given by the (minimal) types of the arguments (and possibly some additional conventions). To replace one type by another in function profiles can lead to a change in function bindings, and the effect can consequently be regarded as *the replacement of one function by another.* We have already argued that coercion-free substitution of functions must be contravariant. But redefinition of functions is necessarily covariant. So, the challenge is to find a safe way to combine covariant redefinition of functions in subtypes with a contravariant subtyping rule for function types. We return to this issue in section 6 after having reported some further complications with higher order functions and subtyping.

4.2 Maximal types and coercion functions

Consider a function with domain T applied to an argument e whose minimal type is a supertype of T. The application $(f\ e)$ may still result in a predictable behaviour if the value of e happens to belong to T. Rather than rejecting the expression as ill-typed, the ABEL type checker can (as one possible option) insert a *coercion* on e into the domain of f: $e\,\mathbf{as}\,T$. The coercion, also called a *retraction*, is a partial function well-defined for the subtype T, and ill-defined on the remaining part of the hierarchy. For 1'st order type families, it is sufficient to define one coercion function for each subtype, having as domain the corresponding maximal type, say $\bar{T}$.

$\hat{}\,\mathbf{as}\,T : \bar{T} \rightarrow T$

This strategy is no longer sound in higher order. The integers *Int* can be divided into three basic subtypes: the strictly positive numbers *Pos*, the strictly negative numbers *Neg*, and the singleton type *Zro* containing only 0. The hierarchy also includes intermediate types *Nat*, *NPos* and *NZro* obtained from taking unions of basic types. Consider a coercion function into, say type $Nat \rightarrow Pos$. The naive generalization from 1'st order results in the following slightly strange:

func $\hat{}\,\mathbf{as}\,Nat{\rightarrow}Pos : (\emptyset \rightarrow Int) \rightarrow Nat \rightarrow Pos$
def $F\ \mathbf{as}\,Nat{\rightarrow}Pos\ \ ==\ \ \mathbf{as}\,Pos \circ F \circ \mathbf{as}\,Nat$

Observe that the expression $(F \circ \mathbf{as}\,Nat)$ inside the function body is ill-typed, since the domain of F is $\emptyset$! To allow every function of a type *related* to $Int \rightarrow Int$ as arguments to $\mathbf{as}\,Nat{\rightarrow}Neg$, however, there seems to be no other choice than to specify $\emptyset \rightarrow Int$ as the domain.

The solution is to define more than one coercion into a given function subtype. For $Nat \rightarrow Pos$ we shall need the following:

func $\hat{}\,\mathbf{as}\,Nat{\rightarrow}Pos : (Nat \rightarrow Int) \rightarrow Nat \rightarrow Pos$
def $F\ \mathbf{as}\,Nat{\rightarrow}Pos\ \ ==\ \ \mathbf{as}\,Pos \circ F$
func $\hat{}\,\mathbf{as}\,Nat{\rightarrow}Pos : (Pos \rightarrow Int) \rightarrow Nat \rightarrow Pos$
def $F\ \mathbf{as}\,Nat{\rightarrow}Pos\ \ ==\ \ \mathbf{as}\,Pos \circ F \circ \mathbf{as}\,Pos$

```
func ^as Nat→Pos : (Zro → Int) → Nat → Pos
def F as Nat→Pos  ==  as Pos ∘ F ∘ as Zro
```

In general we could for a given function type $S \to T$ introduce one coercion function for every supertype of $S \to T$. Some of these are, however, *redundant* and can safely be removed.

Definition 4.3

1. A coercion function is redundant if there exist another coercion function with the same name, the same right hand side (syntactically speaking), and a larger domain.
2. A coercion function is also redundant if every (complete) application is ill-defined.

The overloading of coercion functions is resolved as for ordinary functions: the type checker will choose the coercion function with a minimal domain (with respect to subtyping), under the restriction that the coercion application is well typed. Consider a function application $f\ e$ for $f : (Nat \to Pos) \to T$ and $e : Int \to Int$. Then a coercion from the set above must be inserted: $f(e\,\mathbf{as}\,Nat \to Pos)$. The coercion function with domain $Nat \to Int$ is then chosen, since it can safely accept e as actual parameter and since it has the minimal domain in the coercion set.

Remark 4.4 If only well typed expressions in total functions were to be considered, there would be no need for an empty type. In that case a contravariant hierarchy would not in general have a unique maximal type. For instance, the subtype family over $Int \to Int$ would in that setting have the maximal types $Neg \to Int$, $Zro \to Int$ and $Pos \to Int$. However, with the empty type included, such hierarchies are lattices as in 1'st order.

4.3 Monotonicity of parameterized types

The general subtype rule for 1'st order parameterized types is:

$$\textbf{Monoty:}\quad \frac{T'_1 \preceq T_1 \ldots\ldots\ldots T'_n \preceq T_n}{U\{T'_1,\ldots,T'_n\} \preceq U\{T_1,\ldots,T_n\}}$$

If generators are allowed to take functional parameters it becomes possible to represent “infinite” data structures. Unfortunately, this has the consequence of breaking the general monotonicity rule for parameterized types. A typical example would be the encoding of sets of T values by predicates.

```
type PSet{T} ==
module
  func mkSet: (T → Bool) → PSet
  1-1 genbas mkSet
  func ^∈^: T × PSet → Bool
  def x ∈ (mkSet F) == F x
endmodule
```

Let $f : Int \rightarrow Bool$ and $f' : Nat \rightarrow Bool$. Then $(mkSet\ f)$ represents a set of integers, whereas $(mkSet\ f')$ represents a set of natural numbers. It seems safe to replace the former by the latter. However, a function h taking a formal parameter S of type $PSet\{Int\}$ can extract the functional component of S and apply it to an integer. This is exactly what happens in the definition of set membership. In that case it is unsound to apply h with a value of type $PSet\{Nat\}$. The correct subtype rule for *PSet* is inferred from the domain of the generator *mkSet*. Contravariance gives $Int \rightarrow Bool \preceq Nat \rightarrow Bool$, which implies that $PSet\{Int\} \preceq PSet\{Nat\}$. This is at best counter intuitive.

Observe that the type of $(mkSet\ f)$ does not reveal the functional value inside the term. Contrast this with the standard 1'st order finite sets:

```
type $Set\{T\}$ ==
module
  func $\emptyset : Set,\ add : Set \times T \rightarrow Set$
  genbas $\emptyset, add$
  func $\hat{}\in\hat{} : T \times Set \rightarrow Bool$
  def $x \in s$ == case $s$ of $\emptyset \rightarrow false \mid add(s', y) \rightarrow x = y \vee x \in s'$ fo
  obsbas $\hat{}\in\hat{}$
endmodule
```

The *Set* type is monotonic. The fact that $Set\{Int \rightarrow Nat\} \preceq Set\{Nat \rightarrow Nat\}$ follows from contravariance on function types.

These examples motivate the following restriction: *Generators are 1'st order functions (viewing formal types as 1'st order types).* This does not preclude the instantiation of a formal type as a higher order type.

5 Profile completion for higher order functions

In [DO95] an algorithm for the construction of a "complete" set of profiles for TGI defined functions is described. Such a profile set forms a covariant family of profiles over syntactic subtypes of the function domain, providing opportunities for strengthened typing of function applications. For instance, for a TGI defined addition operator on natural numbers the following profile set may be constructed:

$$\textbf{func}\ \hat{}+\hat{}:\quad \begin{array}{r} Nat \times Nat \rightarrow Nat \\ Pos \times Nat \rightarrow Pos \\ Nat \times Pos \rightarrow Pos \\ Zro \times Zro \rightarrow Zro \\ \oslash \times Nat \rightarrow \oslash \\ Nat \times \oslash \rightarrow \oslash \end{array}$$

where the last two profiles express strictness in both operands.

The theory for profile set construction can be lifted to higher order without much difficulty. It does, however, contain points considered too specialised and space consuming for the present paper. In this connection we therefore refer the reader to a forthcoming paper [Krib].

6 Subtype replacement

In the implementation of 1'st order data types, volume constraints typically have to be applied. Thus, the implementation of a data type T may consist of the following two steps, see [Dah92]:

1. Define a subtype $T' = \{x : T \bullet P(x)\}$, where $P : T \rightarrow \mathit{Bool}$ is a restriction on data volume in some sense. For every function profile $f : D \rightarrow C$ depending on T, a corresponding profile $f' : D' \rightarrow C'$ is introduced, where D' (C') is obtained from D (C) by replacing every occurrence of T by T'. It must be proved that $f' \sqsubseteq (f\,\mathbf{as}\,D' {\rightarrow} C')$. Subtype monotonicity gives $D' {\preceq} D$ and $C' {\preceq} C$. If $D' {\prec} D$ then f' may be a redefinition of f taking advantage of that fact, and if $C' {\prec} C$ then f' may have to be a redefinition, coercing the function value, if necessary.
2. Strongly implement the type T' and the T'-dependent functions taking advantage of the computer hardware.

Remark 6.1 *The notation f' is our way to refer to the redefined version of some function f. In an actual specification the symbol f will be overloaded.*

In our setting the approximation relation and (nonstrict) equality are defined as follows. Let d and e be expressions of the same type, say W. Then:

- $d \sqsubseteq e \;\triangleq\; \mathcal{D}[d] \Rightarrow \mathcal{D}[e] \wedge \forall x : X \bullet (d\,x) \sqsubseteq (e\,x)$, for $W = X \rightarrow Y$, and
- $d \sqsubseteq e \;\triangleq\; \mathcal{D}[d] \Rightarrow \mathcal{D}[e] \Rightarrow \neg(e == \bot) \wedge d = e$, for W 1'st order.
- $d == e \;\triangleq\; d \sqsubseteq e \wedge e \sqsubseteq d$,

where $\mathcal{D}$ is a meta-operator checking the definedness of its argument expression.

In many cases one can obtain equality between the redefined and restricted functions: $f' == (f\,\mathbf{as}\,T' \rightarrow U')$. For instance, that is the case for an automatically redefined equality relation over a one-to-one generator basis (cf. [DO95]). In general, however, intermediate T values could violate the restriction, even if the final function value would not.

The type T' is in itself a (partial) implementation of T, and we may hence wish to "replace" T by T' in some specification S. Call the result S'. Intuitively, this amounts to replacing every occurrence of T in S by T', resulting in a subsequent rebinding of function symbols. However, since T' is a (semantic) subtype of T, the definition of T' depends upon T. Hence S' must at least contain the generators for T, and the functions occurring in the constraining predicate. For that reason, S' will in general contain both T' and T. For simplicity, we assume that the resulting occurrences of T in S' are isolated within the T module.

For a higher order specification, the restricted type T is necessarily 1'st order. However, subtype monotonicity is lost in general, so that the domain and codomain of an affected function are not in general subtypes of the original domain and codomain. Consider for example the *map* function of section 2:

$$\mathbf{func}\ \mathit{map}: (\mathit{Nat} \rightarrow \mathit{Nat}) \rightarrow \mathit{Seq}\{\mathit{Nat}\} \rightarrow \mathit{Seq}\{\mathit{Nat}\}$$

Replacing *Nat* by the subtype *BNat* of section 4.1 yields a function with the domain $\mathit{BNat} \rightarrow \mathit{BNat}$ and the codomain $\mathit{Seq}\{\mathit{BNat}\} \rightarrow \mathit{Seq}\{\mathit{BNat}\}$, neither of which is a subtype of that of *map*.

Loss of monotonicity hence seems to invalidate the simple form of redefinition obtained by using the function body as it is, and merely coercing the function value if necessary. We aim to prove that the simple form for redefinition nevertheless is sufficient, provided we replace *type families* rather than single types. We call this *subtype replacement.*

First of all, it is important to distinguish between "raw" specifications and type checked specifications. The type checker in ABEL is allowed to insert applications of coercions when the type of an expression is a supertype of the expected type (inferred from the context). Given a function $f : T_1 \rightarrow U$ and an expression $e : T_2$, where $T_1 \prec T_2$, the type checker will transform the application into $f(e \,\mathbf{as}\, T_1)$. Thus, in type checked specifications, to which type replacement will be applied, no function is applied outside its domain.

In the context of higher order functions the above requirements to function redefinition must be somewhat modified. For the purpose of the theorem below it is sufficient to require:

- Every function whose *c-codomain* has been restricted must be redefined (or proved with respect to the new profile).

To minimize bureaucracy we shall confuse the distinction between a type expression TE and its associated value set V_{TE}, and also confuse the subtype relation by the corresponding subset relation on the value sets. The following is easily proved:

Lemma 6.2 Let T be the head of a subtype family, and let $P : T \rightarrow \mathit{Bool}$ be a restriction to be applied on every member of the family. Let the restriction by P of each $T_i \preceq T$ be $T_i^{'}$. For an arbitrary type expression U, let U' denote the result of replacing every T_i by $T_i^{'}$ in U. Then $S \preceq U$ implies that $S' \preceq U'$ (or more formally that $V_S \subseteq V_U$).

Proof: By induction on the subtype proof of $S \preceq U$. The leaves in the proof are judgements on the form $C \preceq D$, for C and D non-parameterized, non-functional types for which an explicit subtype relation is declared. If C and D are subtypes of T, then $C' \preceq D'$ follows since the subset relation (on value sets) obviously is maintained under restriction by a predicate. Conversely, if C and D are not subtypes of T, then C' (D') is just C (D), and $C' \preceq D'$ follows by assumption. Now, consider an inference step in the subtype proof of, say $C \preceq D$. Since the outermost type constructor in C' (and D') is the same as in C (and D), $C' \preceq D'$ must be inferred by the same rule as $C \preceq D$. By induction we may assume $C_i' \preceq D_i'$ for every premise $C_i \preceq D_i$ in the original proof, and the claim follows. □

Theorem 6.3 Let S be a specification, possibly higher order, (consisting of type definitions, function profiles, and function definitions). And let e be an arbitrary expression in these functions. Denote by S' and e' the result of replacing every member of the T family with its restriction by some predicate on T (excepting the T module itself). Then S' and e' are well typed, and e' is an approximation to e: $e' \sqsubseteq (e \,\mathbf{as}\, W')$.

Proof: First we prove by induction on e that $e : W$ implies $e' : W'$. So assume $e : W$. There are two different cases to consider:

- e is a variable or a function symbol. Then $e' : W'$ follows from the way type replacement is defined.
- e is an application $e_1\ e_2$, where $e_1 : V_1 \rightarrow W$ and $e_2 : V_2$ and $V_2 \preceq V_1$. e' is the application $e_1'\ e_2'$, where the induction hypothesis gives $e_2 : V_1'$ and $e_1 : V_2'$. It is hence sufficient to prove $V_2' \preceq V_1'$. But this follows from lemma 6.2.

Profiles $f : W'$ are seen to be valid from the requirements on function redefinitions: due to insertion of coercions it can be proved that any complete application of f' can be assigned the c-codomain of f'. $e' \sqsubseteq (e\,\mathbf{as}\,W')$ follows from function monotonicity and the fact that each individual function replacement introduces an approximation. $\square$

Remark 6.4 All constructively defined ABEL functions are monotonic (with respect to definedness). This follows from the facts that generators are strict, and that **if** and **case** constructs are strict in the discriminand. Notice that higher order coercions are implemented in terms of 1'st order coercions.

Remark 6.5 The occurrences of "type expressions" in the names of coercion functions are *not* replaced. Thus the effect of type replacement is just to change function bindings to reflect function redefinitions, and the only "new" coercion applications introduced are the ones occurring inside these redefined functions.

The key to the result is that *every* member of the subtype family is replaced by its restriction, and that *all* functions whose c-codomain is thereby affected are suitably redefined (or re-proved). The redefinitions make it impossible to build values outside the restricted domains, which means that further coercions are not needed.

A simple way to achieve the necessary function replacements is to redefine those generators of T that can produce values outside T' by coercing the function value, and then rebind all occurrences of these generators (and implicitly the occurrences of all affected functions).

In some cases execution efficiency can be gained by applying the original generators, at the expense, however, of proving in each case that coercion is superfluous. For instance, that would be the case for the successor applications building up the integer quotient of bounded integers, and for those occurring in the length function of sequences whose lengths are more strongly restricted.

Remark 6.6 We have restricted our attention to specifications in which all used functions are constructively defined. That seems reasonable in the context of type replacement for executability. In particular, any loosely specified function used in some module can be assumed to be bound to a defined one whose definition satisfies the axioms assumed for the former.

References

[Bar84] H. P. Barendregt. *The lambda calculus: its syntax and semantics.* North-Holland, Amsterdam, 1984.

[Bar92] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of logic in computer science — volume 2*, pages 118–309. Oxford Science Publications, 1992.

[Dah92] O.-J. Dahl. *Verifiable Programming.* International Series in Computer Science. Addison-Wesley, 1992.

[DLO86] O.-J. Dahl, D.F. Langmyhr, and O. Owe. Preliminary report on the specification and programming language ABEL. Research report 106, University of Oslo, December 1986.

[DO95] O.-J. Dahl and O. Owe. On the use of subtypes in ABEL. Research Report 206, University of Oslo, August 1995.

[Gou90] J. A. Gougen. Higher-order functions considered unnecessary for higher-order programming. In D. A. Turner, editor, *Research topics in functional programming*, chapter 12, pages 309–351. Addison-Wesley, 1990.

[HS86] J. R. Hindley and J. P. Seldin, editors. *Introduction to combinators and λ-calculus.* London Mathematical Society Student Series. Cambridge University Press, 1986.

[Hug90] J. Hughes. Why functional programming matters. In D. A. Turner, editor, *Research topics in functional programming*, chapter 2, pages 17–42. Addison-Wesley, 1990.

[Joh85] Th. Johnson. Lambda lifting: transforming programs to recursive equations. *LNCS*, 201:190–203, 1985.

[Kria] B. Kristoffersen. Higher order terminating generator induction. In preparation.

[Krib] B. Kristoffersen. Profile completion for higher order functions. In preparation.

[Mil91] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Owe91] O. Owe. Partial logics reconsidered: a conservative approach. Research Report 155, University of Oslo, June 1991.

[vBF93] S. van Bakel and M. Fernández. Strong normalization of typeable rewrite systems. In *First International Conference on Higher-Order Algebra, Logic, and Term Rewriting '93, LNCS 816*, pages 21–39. Springer-Verlag, 1993.