

Extensible algebraic datatypes through prototypes and subtyping

Petri Mäenpää
Nokia Networks

Kenneth Oksanen
Helsinki University of Technology

Summary

Abstract

This paper presents a new style of functional programming. The prevalent style is to define an algebraic datatype by its constructors, and functions on it by pattern matching. Programming with prototypes, on the other hand, is one of the prevalent styles of object-oriented programming. We fuse these two approaches to strive for the best of both. We propose a new functional language design for algebraic data types. Our proposal replaces the prevalent constructor-based definitional system by a prototype-based one. This is done in a way that does not affect one of the main strengths of functional programming: the use of higher-order functions defined by pattern matching on algebraic data types.

1 Introduction

This paper presents a new style of functional programming. The prevalent style is to define an algebraic datatype by its constructors, and functions on it by pattern matching. Programming with prototypes, on the other hand, is one of the prevalent styles of object-oriented programming (Abadi and Cardelli 1996 provides a lucid and comprehensive exposition).

We fuse these two approaches to strive for the best of both. Principal advantages of object-oriented programming are that code can be reused through inheritance, and that methods (i.e. functions from the point of view of functional programming) can operate polymorphically on related types of data by subtyping and subsumption. Furthermore, methods can be made resilient with respect to changes in the structure of data in a way functional programming has not been able to match so far.

Our contention is that this relative deficiency of functional programming can be remedied without going the full way into the object-oriented realm. We propose an alternative functional language design for algebraic data types instead. Our proposal replaces the prevalent constructor-based definitional system by a prototype-based one. This is done in a way that does not affect one of the main strengths of functional programming: the use of higher-order functions defined by pattern matching on algebraic data types.

The characteristic property of our system is that a data type can be extended by extending its defining prototypes so that the functions defined on this data type work without changes. The prototypes can be extended by new fields and *subprototypes*, which correspond to subclasses in class-based object oriented languages.

The difficulties in extending data and operations on it are well known (e.g. Cook 1990, Krishnamurti et al. 1998). Object-oriented languages allow modular extension of a data type by new variants, but not modular addition of new methods. To do this, one must modify existing classes. Functional languages allow modular addition of new functions, but not of variants to a data type. Krishnamurti et al. (1998) show how to achieve both goals at the same time, but at the price of introducing code that is extraneous to the data types and methods, and a new programming pattern.

Our proposal, on the other hand, does not use any extraneous code. Due to this constraint, our solution is only a partial one. It solves the problem of extending data in a restricted way so that the functions remain resilient, and of defining new functions on the data. The restriction on data extension is that new variants cannot be added to a data type without changing functions on the data type. But existing variants can be extended by new data components and new subprototypes.

This new programming style is not an object-oriented one, because it lacks the notions of self, methods, and late binding. These are an orthogonal issue and could be added as in Objective Caml (Leroy et al. 1999), for instance. We want to study the use of prototypes for defining data in functional programming. This is because we want to isolate a central feature of the object-oriented paradigm and show that its use carries over to the functional paradigm. The use of self and methods introduces a very powerful form of recursion, and makes the type system of a language considerably more complicated. We want to present a style of functional programming that need not take this complication into account. Therefore we do not have methods at all, only functions that operate on data defined by prototypes and subtyping.

Our language design can be seen as based on records just as well as prototypes, because we don't use the notions of self and dynamic dispatch. This places our design clearly in the realm of functional rather than object-oriented programming. However, we use records in a way that appears to be novel.

This approach is realised in the persistent functional programming language Shines presently developed in the joint HiPro project between Nokia Networks and Helsinki University of Technology. Shines is still under construction, but its compiler is already completely written in Shines itself. It runs on the persistent functional data manager Shades (Oksanen 2000). The language is intended for robust large-scale industrial applications, covering at least database pro-

gramming and fault-tolerant, real-time systems. Ordinary functional languages are not adequate for them.

Our approach has been extensively tested in practice in coding the Shines compiler. It has been incrementally extended by new phases, such as optimization. The prototypes that define the algebraic data types of the compiler have been redefined upon each extension, adding new fields. Still, hardly any modifications have been needed in the existing functions that operate on those types.

Shines has strict evaluation, because this is dictated by the functional data management of Shades. Presently known efficient implementations of lazy functional languages are heavily non-functional, using techniques such as graph rewriting. The new functional programming style described in this paper has evolved in the design and implementation process of the Shines language.

Related to Shines in intended applications is the language Erlang developed at Ericsson, but it has a different programming style, deriving fundamentally from Prolog. It diverges somewhat from the prevalent style of functional programming, and also carries heavy influences from evolving in an untyped world.

A data manager Mnesia is implemented on top of Erlang, which costs considerably in efficiency. Efficiency has been perhaps the dominant bottleneck preventing the spread of functional programming into industry. Shades, in contrast, functions as the runtime of Shines, so the overall architecture has two layers instead of three in Erlang / Mnesia. The efficiency of Shines / Shades has proved sufficient for demanding real-time industrial applications (Oksanen 2000).

We present a type system with top-level polymorphism and type inference à la Hindley-Milner, in the tradition of ML. Extensions to it for higher-order and dependent types for e.g. database applications are work in progress. We apply Nordlander's (1998, 1999) algorithm for polymorphic subtype inference to our system. Nordlander's language design for O'Haskell is the work that relates most closely to ours. In our design as well as his, subtyping is name-based rather than structural. This is common in object-oriented programming languages (e.g. Java), although research has focused on structural subtyping.

Due to the similarities between Nordlander's system and ours, his polymorphic subtype inference algorithm applies directly to our system. Its main characteristic is that it never yields a type with subtype constraints, which is in contrast to other algorithms. We have found it pragmatically very useful due to this. Theoretical study has focused on algorithms that find the most general possible type, which typically contains subtype constraints. This generality, however, makes these algorithms pragmatically unappealing, because the user cannot easily deal with the typically overwhelming amount of constraints returned. Nordlander's algorithm is incomplete, i.e. it is not always able to infer a type at all. But these occasions are very rare — usually the type inference works just as well as ordinary Hindley-Milner type inference, which it conservatively extends. On the rare occasions when the algorithm is not able to infer a type, or the type is not as general as intended by the programmer, he can guide the algorithm by a type annotation.

Our prototype (or record) extension mechanism appears to be new. It defines subtyping by multiple inheritance in object-oriented terms, but also in a way that is conditioned by subtyping. This seems to give a new language design and record calculus for multiple inheritance.

The plan of this paper is as follows. We first describe

the standard way of defining algebraic data types (Section 2). Then we present our proposal in terms of prototypes (Section 3). Section 4 describes a technique for defining functions by pattern matching on our algebraic data types, which combines pattern matching with subtyping in a novel way. New values are generated from prototypes by cloning, as in prototype-based object-oriented languages (Section 5). Section 6 shows how multiple inheritance is used for the incremental definition of data types. A fairly substantial programming example is presented in Section 7. Section 8 presents the formal rules underlying our proposed language design, and Section 9 concludes and discusses related work.

2 Algebraic data types defined by constructors

Our presentation proceeds in terms of the running example of the algebraic data type of binary trees. The standard way of defining them is by the constructors `Nil` and `Branch`, in Haskell notation

```
data Tree t = Nil | Branch t (Tree t) (Tree t)
```

Now suppose we have defined a function by pattern matching over the constructors of `Tree`, for counting the inner nodes of a tree.

```
count Nil = 0
count (Branch a t1 t2) =
  1 + (count t1) + (count t2)
```

In everyday programming practice, the need frequently arises to change the definition of an algebraic data type. Thus we might want to add an additional piece of information of type `t` to each inner node of a tree. We would redefine trees as

```
data Tree t = Nil |
  Branch t t (Tree t) (Tree t)
```

Notice that this definitional system forces modifying the whole definition of `Tree`, because a `Branch` has no independent definitional status. A facility for local modifications of constructors would be desirable for enhancing modularity. Our proposal will address this issue.

This redefinition of `Branch` is a minimal change, but it has consequences for all functions that operate on binary trees. The `Branch` case of each such function must be recoded. In the case of `count`, the code must be modified to

```
count Nil = 0
count (Branch a b t1 t2) =
  1 + (count t1) + (count t2)
```

It is a nuisance to recode functions upon redefining algebraic data types that they operate on, when the redefinitions have no semantic bearing on the functions. This is semantically superfluous manipulation of syntax, forced by the way

algebraic data types are represented in standard functional languages. The main reason is that the components of a constructed value are identified by absolute position rather than by name.

Object-oriented programmers are used to avoiding this nuisance by object abstraction. But this way of avoiding the problem is trivial in the absence of algebraic data types with pattern matching. It is not a unique property of the object-oriented paradigm, but can be achieved just as well by means of abstract data types in the functional paradigm.

Still, algebraic data types make the maintenance and evolution of functional code tedious and error-prone. The object-oriented paradigm has been so successful in part because of the resilience of code in the face of such changes in data. Do the object-oriented ideas carry over to the functional paradigm? Here is our proposal for achieving this.

3 Extensible algebraic data types through prototypes

Instead of defining a data type by its constructors, consider defining it by prototypes. For example, our original binary tree type (`Tree t`) is defined in Shines as follows.

```
Tree(A)    := <>;
Nil(A)     := EXTEND Tree(A);
Branch(A)  := EXTEND Tree(A) WITH
  <key := A, left := Tree(A), right := Tree(A)>;
```

Here `A` is a type variable that serves the purpose of `t` in the Haskell version. Three prototypes are defined in an order of inheritance, using records of the form `< ..., l := A, ... >`. In this form, `l` is a field label and `A` is a type whose *base prototype* (abstract class in the terminology of class-based object-oriented languages) instantiates field `l` to a default value. A default value can also be some other constant expression besides a prototype. In that case the field must be annotated by a type, unless type inference can derive it. For example, the `key` field could be instantiated by `key := 0 :: INT`, which can be abbreviated to `key := 0` due to type inference.

The base prototype of binary trees is `Tree(A)`, which simultaneously defines the type `Tree(A)`. The prototype is defined as the empty record, and the type correlatively becomes defined as the empty record type. Thus a prototype and its type share the same name. This overloading serves to make definitions of algebraic data types more concise. We seek to maximize the role of type inference so that the programmer needs to write type expressions as seldom as possible. In the case of prototype objects, the type can be immediately inferred from the name of the object. Prototypes cannot be referred to without name, so they are not lightweight, to use a term current for records. Moreover, the syntactic context always disambiguates whether a type or a prototype is meant.

The `EXTEND WITH` operation is for inheritance through *prototype extension*, which is a form record extension that induces a name-based subtyping relation on prototypes. One may omit the `WITH` part, as in the definition of `Nil`. The prototypes `Nil(A)` and `Branch(A)` inherit the fields of the prototype `Tree(A)` and their default values. The prototype `Tree(A)` has no fields, but there may of course be some in the general case. Furthermore, the subtyping relationships

```
Nil(A) <: Tree(A)
Branch(A) <: Tree(A)
```

become declared.

Letting data type constructors be subtype-related separate definitions opens up a modularity boundary within data types, which allows important opportunities for extending the data type.

Prototype extension uses overriding. If two fields have the same name in an extended prototype, say `l := A` and `l := B`, then `l := B` overrides `l := A` if `B <: A`. (Here `A` and `B` are names of prototypes as well as the respective types.) That is, we choose the smaller of the alternative types of name-conflicting fields with no regard for their order of occurrence. If neither `A <: B` nor `B <: A` holds, an error results. This rule is also used to resolve field name conflicts in multiple inheritance. Section 6 discusses the issue further and justifies the rule.

A prototype must be defined before its use in other prototype definitions. This condition rules out cyclic systems of prototype definitions.

The difference of our proposal to standard accounts of functional programming languages is the use of records, with inheritance and subtyping, instead of variants to define algebraic data types. One can even use multiple inheritance (Section 6). Our system for records and subtyping on them is very similar to that of Nordlander (1998, 1999) except that the rules for record extension and concatenation differ. Another difference is in functions on records. We define them by name-based matching, which is essential for the kind of resilience of functions with respect to data type extension that we propose.

Due to multiple inheritance and subtyping, our system provides a more general form of definition of algebraic data types than the standard one based on constructors.

Haskell allows defining data types with field labels (Report on the Programming Language Haskell 98, section 3.15), which are related to our present proposal. However, Haskell takes the constructor-based approach to algebraic data types as primitive, and the one with field-labels is translated back to it. We do not need constructors, Haskell-like data types or variant types at all. Instead, we take record types as primitive, and build prototypes with declared subtyping on their basis. The main difference is that Haskell records do not enjoy subtyping. Records in O'Haskell (Nordlander 1998, 1999) do.

A further difference to Haskell's data types with field labels is their lack of default values. Our prototype fields have them — it is in the nature of prototypes. This makes the use of data types with field labels considerably more natural. For instance, Haskell assigns the undefined value if no other value is determined. Our approach never leads to this semantically poorly motivated situation. To take an example from the Haskell report, given the data type definition

```
data T = C1 {f1,f2 :: Int}
       | C2 {f1 :: Int,
            f3,f4 :: Char}
```

the update expression `C1 f1 = 3` means the same as `C1 3 undefined`. In the corresponding Shines definition, all fields

must be supplied with a default value.

```
T := <>;
C1 := EXTEND T WITH
    <f1 := 0 :: INT,
    f2 := 0 :: INT>;
C2 := EXTEND T WITH
    <f1 := 0 :: INT,
    f3 := ' ' :: CHAR,
    f4 := ' ' :: CHAR>;
```

The corresponding update expression `C1.<f1 := 3>` means the same as `C1.<f1 := 3, f2 := 0>`, so Haskell's semantical malformation is avoided by using prototypes. The above Shines data type definition also illustrates the use of the primitive types `INT` and `CHAR` as field types. Primitive types are not prototypes, so the default values `0` and `' '` must be specified explicitly. (All of the type annotations in this example could have been omitted in fact, because type inference can derive them.)

Returning now to our tree example, the expressions `Tree(A)`, `Nil(A)` and `Branch(A)` conceived of as types are names of record types, as follows.

```
Tree(A)   = <>
Nil(A)    = <>
Branch(A) = <key :: A,
           left :: Tree(A),
           right :: Tree(A)>
```

In virtue of subsumption and the induced subtyping `Nil(A) <: Tree(A)` and `Branch(A) <: Tree(A)`, we also have

```
Nil(A) :: Tree(A)
Branch(A) :: Tree(A)
```

The expressions `Nil(A)` and `Branch(A)` stand for values here, whereas `Tree(A)` stands for a type. This is because of the overloading of prototype names: they can stand for values as well as types. Declared subtyping between variants of a data type appears also in O'Haskell (Nordlander 1998, 1999), but O'Haskell specifies data types in other respects like Haskell, i.e. by variant types rather than record types.

Our approach makes use of subtyping to make the type hierarchy more fine-grained than in functional languages without subtyping. Nothing is lost, because of subsumption. Yet this provides more flexibility and opportunities for code reuse. Because standard functional languages lack subtyping, our approach is blocked in them. The main reason for the lack is that Hindley-Milner type inference is hard to fit together with polymorphic subtyping.

Objective Caml opts for explicit coercions instead of (implicit) name-based subtyping as found in Shines, O'Haskell or Java. Java lacks (parametrically) polymorphic subtyping, on the other hand.

Obviously, we can have neither structural subtyping nor structural type equality, because nothing distinguishes between e.g. `Tree(A)` and `Nil(A)` on structural grounds. Instead, our subtype and type equality relations are induced by inheritance declarations, supplemented with the obvious general rules for subtyping (reflexivity, transitivity) and type equality (reflexivity, symmetry, transitivity).

4 Name-based pattern matching with subtyping

A main source of the resilience of functions in extending data is that we combine name-based declared subtyping with name-based matching that supports subtyping. The `count` function is defined in Shines as

```
count(t) :=
  MATCH t
  ON Nil DO 0
  ON Branch.<t1 := left, t2 := right> DO
    1 + count(t1) + count(t2)
  END
```

The key field is not mentioned, which is satisfactory for several reasons. First, it is irrelevant for `count`. Second, adding another field that is irrelevant for counting, like `key2` in our example, prompts no changes in the code. It remains resilient to such data extensions. (We assume that `t` contains no occurrences of the base prototype `Tree(A)`, i.e. that all nodes are of the form `Nil` or `Branch`. An otherwise-arm could be added to the match to handle this default case, but this is left to the responsibility of the programmer.)

Name-based pattern matching on prototypes (or records) is essential for achieving this resilience. Pattern matching on ordinary datatypes as in SML and Haskell fails to be resilient in this respect, because they require an exhaustive list of arguments to constructors. Name-based pattern matching on records requires mentioning just the fields that are used in the right hand side of the definition, not an exhaustive list. Our design thus provides a certain amount of width extensibility of data by allowing new fields to be added to a variant of a data type without necessitating the modification of functions on the data that don't use the new data.

Our design also provides extensibility in the depth dimension in the form of subprototyping. A `MATCH` expression need not provide an exhaustive list of variants for the scrutinized expression. It suffices that all variants of the data type can be subsumed into the arms. Subtyping enables considerably more robust and resilient pattern matches than in functional programming languages without it.

Not all extensions to data are possible without modifying functions, however. Adding a new variant to a data type still forces the programmer to modify existing functions. Krishnamurti et al. (1998) get around this difficulty as well, but at the price of introducing extraneous components to the data, to accommodate for possible future extensions by new variants. We use weaker means, and achieve somewhat weaker results correspondingly.

In our prototype-based version, the extended definition of `branch` is

```
Branch(A) := EXTEND Tree(A) WITH
    <key := A,
    key2 := A,
    left := Tree(A),
    right := Tree(A)>
```

where the new field `key2` is of type `A`. The old code for `count` is resilient. It works without changes with the modified definition of trees.

One might argue that the same could be achieved by instantiating the type variable `A` to another value, a pair of values of type `A`, making use of parametric polymorphism. But the problem is more general. For example, we might just as well want to add a third subtree to each branch of a tree for some other purpose, converting binary trees into ternary ones. No instance of `A` would account for this situation in a natural way. Moreover, even if a certain modification to trees could be avoided by polymorphism, we do not want to press the programmer to unnatural encodings that may ensue thus in general.

To make the data type extension fully modular, we could define a subprototype `NewBranch` of `Branch` instead of modifying the code for `Branch`.

```
NewBranch(A) := EXTEND Branch(A) WITH <key2 := A>
```

The function `count` would still be resilient to the extension. This is because the new declaration implies that `NewBranch` is a subtype of `Branch`. The arm of `count` for `Branch` also works for `NewBranch` as such, due to the combination of subtyping and name-based pattern matching on records. A `NewBranch` expression matches a `Branch` pattern in virtue of subsumption.

This makes the order of patterns crucial. The first pattern that matches is chosen as is usual in functional languages, but the programmer also needs to take into account that the match may be affected by subsumption. Thus if a match expression has an arm for `Branch` as well as for `NewBranch`, the `NewBranch` arm must be placed first in order to be reachable. If a base prototype is used as a pattern, it is typically placed in the last arm, because all values of the scrutinized type would match it. So it functions as an otherwise-case.

This kind of modular extension of datatypes in the depth dimension, by new subprototypes, is characteristic of object-oriented languages. It allows a form of stepwise data refinement that is lacking in other kinds of languages.

We are not aware of other functional languages besides Shines that have this kind of design for algebraic data types and functions on them. O'Haskell (Nordlander 1998, 1999) comes closest, because it has datatypes with name-based subtyping and stepwise data refinement. But it does not support the resilience of functions on data types in the present kind of situation where a datatype is extended by new fields.

5 Cloning values of algebraic data types

As in prototype-based object-oriented languages, new objects or values are created by *cloning* prototypes. Cloning a prototype `P` is just assigning it as a value to a variable `v`,

```
v := P.< ..., l := b, ... >
```

possibly updating some of its fields to new values by bindings of the form `l := b`. Here `l` is the name of a field and `b` is a value of appropriate type. When cloning a prototype, any number of its fields may be updated. Thus we might create a new binary tree value `br1` from the `Branch(A)` prototype, by updating its `key` field to a new value `0` instead of the default value `A` in the definition.

```
br1 := Branch(0)
```

Now `Branch(0)` means the same as `Branch(INT).<key := 0>` and has type `Branch(INT)`. The parameter `A` is thus overloaded in the same way as prototype names are. It signifies the type parameter of the prototype, but it also signifies the default value of that type, which is used in the instantiation. This kind of overloading yields a pleasantly succinct notation: we avoid writing the considerably more verbose expression

```
br1 := Branch(INT).<key := 0>
```

If the default values supplied at polymorphic prototype instantiation are constant expressions, they can be evaluated at compile time. But also non-constant expressions are allowed. Their values are not determined until run time. Thus we might define a function

```
new_branch(n) := Branch(n)
```

for constructing a branch with a given key `n`.

6 Multiple inheritance

In contrast to the standard approach to algebraic data types in functional programming, our approach allows definition by multiple inheritance. A new prototype may be defined by inheritance from several distinct algebraic data types.

Multiple inheritance is just simple prototype (or record) concatenation unless field names conflict. The conflict resolution rule is the same as with prototype extension. Our rule is illustrated by the general multiple inheritance situation

```
Proto1 := < ..., l := A, ...>;
Proto2 := < ..., l := B, ...>;
Proto := EXTEND Proto1, Proto2 WITH <...>;
```

Now `Proto` is defined as the concatenation of the prototypes `Proto1` and `Proto2` and an optional `WITH` part, which we now assume not to contain the field `l`. The type of field `l` in `Proto` is `A` if `A <: B`, and `B` if `B <: A`. Otherwise an error results, because the prototypes `Proto1` and `Proto2` have no unique greatest common subtype. Any number of prototypes and new fields may be concatenated by the `EXTEND` operator in a prototype definition.

Our rules for prototype extension (simple inheritance) and concatenation (multiple inheritance) differ from those of Nordlander (1998, 1999) for records, although in many other respects our design is close to his.

The prototype concatenation rule is justified as follows. We regard algebraic data types as restrictions on values of prototypes. Inheritance aims at reusing code, including type information. Maximal reuse is achieved by choosing the greatest common subtype of prototypes that are inherited, if such a type exists. This is because that type includes all the restrictions imposed on all fields of the parent prototypes, but nothing more.

Multiple inheritance induces multiple subtyping, so `Proto <: Proto1` and `Proto <: Proto2`. It is these induced subtype relationships that require the stated conditions on greatest common subtypes, in order for multiple inheritance to be sound.

Here is an example from actual code of the Shines compiler. It contains the syntactic categories `Expr` (expressions), `BinOp` (binary operations), `BoolExpr` (boolean expressions), and `And` (boolean conjunctions), represented by the prototypes (omitting parts that are irrelevant in this example):

```
Type      := <...>;
BoolType  := EXTEND Type WITH <...>;
Expr      := <type := Type, ...>;
BinOp     := EXTEND Expr WITH
  <left := Expr, right := Expr, ...>;
BoolExpr  := EXTEND Expr WITH <type := BoolType>;
And       := EXTEND BinOp, BoolExpr;
```

The first two definitions determine that `BoolType` inherits from `Type`. Now `BinOp` inherits the value `Type` for its `type` field from `Expr`, whereas `BoolExpr` overrides this value by `BoolType`. This is because `BoolType` is a subtype of `Type`. Finally, `And` multiply inherits from `BinOp` and `BoolExpr` so that it inherits a `left` field and a `right` field from `BinOp`, but a `type` field from `BoolExpr`.

This is an example of the utility of our rule of multiple inheritance for the programmer. If any rule based on order of the multiply inherited prototypes were used, the programmer would not be able to inherit some fields from one parent, and other fields from another parent in a situation where the names of the inherited fields conflict.

For situations where the programmer wants to explicitly override a field defined in the parents, he may force a field to have a certain type. Continuing our example involving `Proto1` and `Proto2`, he may define

```
Proto := EXTEND Proto1, Proto2 WITH <l := C>
```

to force field `l` to have type `C` in `Proto`. This requires that `C <: A` and `C <: B`, so that the induced subtype relationships `Proto <: Proto1` and `Proto <: Proto2` become sound. If these conditions do not hold, an error results.

7 Programming example

We now give a more substantial programming example in Shines. It is a fragment of code from the Shines compiler that folds constants.

```
Expr := <folded? := FALSE>;

Var := EXTEND Expr WITH <name := "x">;

Plus := EXTEND Expr WITH
  <left := Expr, right := Expr>;

Constant := EXTEND Expr WITH <value := 0>;

Int := EXTEND Constant;

Symbol := EXTEND Constant WITH <name := "Pi">;
```

```
cf(x) :=
  IF x.folded? THEN x
  ELSE do_cf(rec_cf(x)).<folded? := TRUE>;

rec_cf(x) :=
  MATCH x
  ON Plus DO
    x.<left := cf(x.left),
      right := cf(x.right)>
  OTHERWISE x
  END;

do_cf(x) :=
  MATCH x
  ON Plus.<left = Int.<i := value>,
    right = Int.<j := value>> DO
    Int.<value := i + j>
  ON Plus.<r := right = Int, l := left> DO
    x.<left := r, right := l>
  ON Plus.<left = Int.<value = 0>,
    r := right> DO
    r
  ON Plus.<left = Int.<i := value>,
    right = Plus.<left = Int.<j := value>,
      rr := right>> DO
    Plus.<left := Int.<value := i+j>,
      right := rr>
  ON Plus.<left = Plus.<lli := left = Int,
    lr := right>,
    r := right> DO
    Plus.<left := lli,
      right := Plus.<left := lr,
        right := r>>
  ON Plus.<l := left,
    right = Plus.<rli := left = Int,
      rr := right>> DO
    Plus.<left := rli,
      right := Plus.<left := l,
        right := rr>>
  OTHERWISE x
  END;
```

Some explanations are in order on the patterns used here. The `=` symbol means equality testing, whereas the symbol `:=` signifies binding of a value to a variable. Thus the first arm of the match in `do_cf` tests if the expression is cloned from the `Plus` prototype or its descendants; and if so, whether its `left` and `right` fields are cloned from the `Int` prototype or its descendants; and if so, the values of the respective `value` fields are bound to the variables `i` and `j`.

The above functions perform constant folding in an expression by transforming the expression recursively. Function `do_cf(x)` matches the expression to a pattern and performs a transformation according to the pattern.

Below is the list of patterns used in `do_cf()`. The first, third, and fourth rule perform the actual addition of constants, the other rules move integer constants together upwards and leftwards in the syntax tree, and the last rule is the default when no constant folding can be done.

1. $int + int \rightarrow int$
2. $x + int \rightarrow int + x$
3. $0 + x \rightarrow x$

4. $int + (int + x) \longrightarrow int + x$
5. $(int + x) + y \longrightarrow int + (x + y)$
6. $x + (int + y) \longrightarrow int + (x + y)$
7. $x \longrightarrow x$

8 Formal syntax and typing rules

Now we present the formal syntax and typing rules of our language design. It is an extension of the standard Hindley-Milner type system for prototypes. Prototypes are just named records, so our rules are for a system of functions and records. The literature abounds with similar calculi, so we do not provide lengthy explanations or arguments. There is very little new on the formal level: only the rules for record extension seem to be without precedent in the literature. The subtyping relation for records is name-based rather than structural, which stands in contrast to the majority of works on subtyping records.

Our main contribution is not so much in these formal rules, but in the way named records and name-based subtyping are used for defining extensible algebraic datatypes and functions by pattern matching on them. Therefore we refer to Nordlander (1998, 1999) for the formal properties of the calculus and other further information regarding the rules. We directly apply his results and his polymorphic subtype inference algorithm in our work.

We omit the treatment of constant terms and primitive constant types for brevity.

8.1 Term syntax

e	$::=$	x	variable
		$e(e')$	application
		$\lambda(x)e$	abstraction
		$e.l$	projection
		$LET\ x\ :=\ e\ IN\ e'$	local definition
		P	prototype name
		$P(e_1, \dots, e_n)$	instance of polymorphic prototype
		$e.<l := e'\rangle$	update
		r	record
r	$::=$	$\langle l_1 := e_1 \{:: T_1\},$ $\dots,$ $l_n := e_n \{:: T_n\} \rangle$	
P	$::=$	id	identifier

The curly braces indicate optional constituents.

Repeated application, abstraction, local definition and update can be defined straightforwardly from these rules.

8.2 Type syntax

T, U, V	$::=$	X	variable
		P	prototype name
		$P(X_1, \dots, X_n)$	polymorphic prototype
		$\langle l_1 :: T_1,$	record type

		$\dots,$	
		$l_n :: T_n \rangle$	
S	$::=$	$(forall\ X_1,$ $\dots,$ $X_n)T$	type scheme
E	$::=$	$x_1 :: S_1,$ $\dots,$ $x_n :: S_n$	type environment

8.3 Subtyping rules

We present the subtyping rules before the typing rules, because the rules for record extension presuppose them.

$T <: T$	(REFL)
$T <: U$	$U <: V$
----- (TRANS)	
$T <: V$	
$A_i <: B_i$	$B_j <: A_j$
(for all i in $cov(P)$)	(for all j in $contrav(P)$)
----- (DEPTH)	
$P(A_1, \dots, A_n) <: P(B_1, \dots, B_n)$	

Here the sets $cov(P)$ and $contrav(P)$ contain the indices for covariant and contravariant argument places of P , respectively. These can be inferred from the prototype declarations exactly the way Nordlander (1998, 1999) does for record types. We furthermore adopt Nordlander's rule

$T <: U$	
----- (CONST)	
$sub(T) <: sub(U)$	

for instantiating subtyping axioms derived from prototype declarations with a substitution sub .

And finally, there is a rule for subsumption.

$E \vdash e :: T$	$T <: U$
----- (SUB)	
$E \vdash e :: U$	

8.4 Typing rules

$E, x :: S \vdash x :: S$	(VAR)
---------------------------	-------

The notation $E, x :: S$ here means the same as the union $E \cup \{x :: S\}$.

$E \vdash e :: T \rightarrow U$	$E \vdash e' :: T$
----- (APP)	

$$E \mid- e(e') : U$$

$$\frac{E, x :: T \mid- e :: U}{E \mid- \lambda(x)e :: T \rightarrow U} \quad (\text{ABS})$$

$$\frac{E \mid- e :: S \quad E, x :: S \mid- e' :: T}{E \mid- \text{LET } x := e \text{ IN } e' :: T} \quad (\text{LET})$$

$$\frac{E \mid- e :: T \quad (X_1, \dots, X_n \text{ not in } \text{fv}(E))}{E \mid- e :: (\text{forall } X_1, \dots, X_n)T} \quad (\text{GEN})$$

Here $\text{fv}(E)$ means the set of free type variables in E .

$$\frac{E \mid- e :: (\text{forall } X_1, \dots, X_n)T}{E \mid- e :: [U_1, \dots, U_n / X_1, \dots, X_n]T} \quad (\text{INST})$$

Then come the rules for prototypes or records. First, there is a rule for prototype or record construction.

$$\frac{E \mid- e_1 :: T_1 \quad \dots \quad E \mid- e_n :: T_n \quad (\text{PROTO})}{E \mid- \langle l_1 := e_1 :: T_1, \dots, l_n := e_n :: T_n \rangle :: \langle l_1 :: T_1, \dots, l_n :: T_n \rangle}$$

Here the field labels l_1, \dots, l_n must be distinct. The order of fields is not significant.

There are three rules of record extension. The first one allows extension in the simple case with no field name conflicts.

$$\frac{E \mid- r :: \langle l_1 :: T_1, \dots, l_n :: T_n \rangle \quad E \mid- e :: T}{E \mid- r ++ \langle l := e :: T \rangle :: \langle l_1 :: T_1, \dots, l_n :: T_n, l :: T \rangle} \quad (\text{EXTEND1})$$

Here the label l must not occur in $\{l_1, \dots, l_n\}$. Associated to this rule, there is an equality rule for computing the value of the extension:

$$\begin{aligned} &\langle l_1 := e_1 :: T_1, \dots, l_n := e_n :: T_n \rangle ++ \\ &\langle l := e :: T \rangle \\ &= \\ &\langle l_1 := e_1 :: T_1, \dots, l_n := e_n :: T_n, l := e :: T \rangle \end{aligned}$$

The second rule is for overriding a field with one of smaller (or equal) type.

$$E \mid- r = \langle l_1 := e_1 :: T_1, \dots, l_n := e_n :: T_n \rangle :: \langle l_1 :: T_1, \dots, l_n :: T_n \rangle$$

$$E \mid- e' :: U \quad U <: T_i \quad (i \text{ in } \{1..n\})$$

$$\frac{E \mid- r ++ \langle l_i := e' :: U \rangle :: \langle l_1 :: T_1, \dots, l_i :: U, \dots, l_n :: T_n \rangle}{\text{(EXTEND2)}}$$

This rule has an associated equality rule for computing the value of the record formed by extension:

$$\begin{aligned} &r ++ \langle l_i := e' :: U \rangle \\ &= \\ &\langle l_1 := e_1 :: T_1, \\ &\quad \dots, \\ &\quad l_i := e' :: U, \\ &\quad \dots, \\ &\quad l_n := e_n :: T_n \rangle \end{aligned}$$

Notice the premise that the type of the extending field must be a subtype of the overridden field. This is called method specialization on override in object-oriented languages (see Abadi and Cardelli 1996, sec. 2.7). Also the type of the resulting prototype is specialized, which can be compared to what Abadi and Cardelli call the silent specialization of the type of self. (Overriding means the same as updating.)

In contrast to the calculus of Abadi and Cardelli, our system allows the type of fields to be specialized on override always, whether they are functions or not. Class-based object-oriented languages generally do not allow field specialization but allow method specialization, because fields can be updated but methods cannot. Abadi and Cardelli allow also methods to be updated, so neither field or method specialization is allowed on override in their calculus. Their calculus is based on prototypes rather than classes, like ours. Abandoning self and late binding gives more generality for us.

The third extension rule accounts for the case where the extending field has larger type than the corresponding field in the record to be extended. In this case, the extending field does not override the inherited one.

$$\frac{E \mid- r :: \langle l_1 :: T_1, \dots, l_n :: T_n \rangle \quad E \mid- e' :: U \quad T_i <: U \quad (i \text{ in } \{1..n\})}{E \mid- r ++ \langle l_i := e' :: U \rangle :: \langle l_1 :: T_1, \dots, l_n :: T_n \rangle} \quad (\text{EXTEND3})$$

Also this extension rule has an associated equality rule.

$$r ++ \langle l_i := e' :: U \rangle = r$$

One may wonder why the last extension rule is needed at all, because it seems to do nothing. It is needed for defining a form of record concatenation from record extension that allows fields with conflicting names to be inherited from different parents, in a way that the field with the smaller type is always inherited.

These rules for record extension seem to be a novelty, in that they have a subtyping condition on the extension that allows the rich form of multiple inheritance described in Section 6.

Prototypes also have the following standard projection rule, which selects a field.

$$\frac{E \mid- r :: \langle l1 :: T1, \dots, ln :: Tn \rangle \quad (i \text{ in } \{i..n\})}{E \mid- r.li :: Ti} \quad (\text{PROJ})$$

Projection has the associated equality rule:

$$\langle l1 := e1 :: T1, \dots, ln := en :: Tn \rangle.li := ei$$

As usual in functional programming, the fields of our prototypes may be functions. In contrast to object-oriented programming, where method extraction is not sound (Abadi and Cardelli 1996), function fields can be soundly extracted from our prototypes by ordinary record projection. Again, our lack of self and late binding provide some additional generality compared to object-oriented languages. Our prototype projection rule thus covers both field selection and method invocation in object oriented terms.

Updating a prototype follows the rule:

$$\frac{E \mid- r = \langle l1 := e1 :: T1, \dots, ln := en :: Tn \rangle :: \langle l1 :: T1, \dots, ln :: Tn \rangle \quad E \mid- e' :: U \quad U <: Ti \quad (i \text{ in } \{i..n\})}{E \mid- r.<li := e'\rangle :: \langle l1 :: T1, \dots, li :: U, \dots, ln :: Tn \rangle} \quad (\text{UPDATE})$$

Note that this rule is almost identical to the rule (EXTEND2). However, update cannot be reduced to extension, because extension is type correct even if the type of e' is greater than the type of li . The rule (EXTEND3) just does nothing in that case, but the corresponding update attempt must not be type correct.

Associated with the update rule is an equality rule that determines the value of the updated record.

$$\begin{aligned} & e.<li := e'\rangle \\ & = \\ & \langle l1 := e1 :: T1, \\ & \quad \dots, \\ & \quad li := e' :: Ti, \\ & \quad \dots, \\ & \quad ln := en :: Tn \rangle \end{aligned}$$

Match expressions are compiled to decision trees as usual in functional languages, so we do not give primitive rules for them.

8.5 Prototype definitions

The concrete Shines syntax

$$T ::= \text{EXTEND } T1, \dots, Tm \\ \{ \text{WITH } \langle l1 := e1 \{:: U1\}, \\ \quad \dots, \\ \quad ln := en \{:: Un\} \}$$

corresponds to the rule syntax

$$T := T1 ++ \dots ++ Tm ++ \\ \langle l1 := e1 :: U1, ln := en :: Un \rangle$$

and declares the subtype relationships

$$T <: T1, \\ \dots, \\ T <: Tm$$

We overload the symbol $++$ by using it for both extension and concatenation, because extension is concatenation with a record of one field. Concatenation can be defined straightforwardly in terms of extension in our system:

$$\begin{aligned} r ++ \langle l1 := e1 :: T1, \dots, ln := en :: Tn \rangle \\ = \\ (\dots(r ++ \langle l1 := e1 :: T1 \rangle) \\ ++ \dots \\ ++ \langle ln := en :: Tn \rangle) \end{aligned}$$

Concatenation is symmetric: $r1 ++ r2 = r2 ++ r1$.

The typing rules are relative to a subtyping theory in the sense of Nordlander (1998, 1999), which is the reflexive and transitive closure of the subtyping axioms induced by the prototype declarations. The record labels in Nordlander's system (and in Haskell) have global scope, which consumes name space in a nonsatisfactory way. Our field names, in contrast, are relative to the name of the prototype in whose declaration they appear.

9 Conclusion and related work

We have presented a design for functional languages that incorporates some features of object-oriented languages to support code extension. Comparisons were made primarily to O'Haskell and Haskell. One could also compare to object-oriented languages, but our proposal is in the domain of functional languages because it does not employ the devices of self or dynamic dispatch.

Jones and Peyton Jones (1999) propose an alternative record system to that of Haskell 98, based on lightweight extensible records. Our records are not lightweight but declared, because their point is to support name-based matching that lends our design the resilience of functions with respect to data extension. Our records are extensible as well, but in a more general manner than those of Jones and Peyton-Jones, because our records allow concatenation rather than mere extension. Furthermore, the concatenation is based on subtyping, and induces subtype declarations, in contrast to the system of Jones and Peyton-Jones.

In some languages (e.g. Objective Caml), a form of record polymorphism is achieved by means of row variables (see e.g. Remy 1991), but row polymorphism is different from subtype polymorphism proper. O'Haskell has subtype polymorphism, but it has no pattern matching for records that supports subtyping. Row polymorphism is closely related to subtype polymorphism, and more specifically structural subtype polymorphism. This is because matches are done on the basis of fields that at least must be present in the scrutinized record expression, rather than on the basis of declared record names. Our approach, in contrast, uses name-based matching and subtyping.

Jones and Peyton-Jones (1999) also use row polymorphism for records. Our proposal, on the other hand, uses ordinary declared subtype polymorphism and pattern matching that supports it. Row polymorphism and the Extensible Visitor design pattern introduced by Krishnamurti et al. (1998) share a certain technique: A form of polymorphism is achieved by introducing variables that may later be used to extend a data structure by instantiation. A related notion of extending data was alluded to in Section 4 above, in the (unsatisfactory) idea of extending data of a polymorphic type by instantiating a polymorphic variable appropriately. Adding extra variables in data to support future extensions introduces an extraneous element to the data. We opt for the kind of subtype polymorphism used e.g. in Java.

Objective Caml uses explicit coercions instead of (implicit) subtyping, and does not relate subtyping to inheritance. Our design is like O'Haskell and Java in basing subtyping on inheritance. This makes subtyping much less cumbersome to use.

Our proposal aims at achieving as much modular code extensibility as possible without anything extraneous to the data and functions on it that we are interested in. We have sought to fuse into a coherent design the kind of subtype polymorphism offered by object-oriented languages like Java, and the kind of parametric polymorphism and higher-order functions defined by pattern matching on algebraic data types offered by SML and Haskell.

References

- [1] Abadi, Martin and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, Heidelberg, 1996.
- [2] Cook, William R. Object-oriented programming versus abstract data types. *Foundations of Object-Oriented Languages*, June 1990, pages 151–178.
- [3] *Report on the Programming Language Haskell 98*. Peyton Jones, Simon et al. February 1999.
- [4] Jones, Mark and Simon Peyton Jones. Lightweight extensible records for Haskell. In Erik Meijer (ed.), *Proceedings of the Haskell 1999 Workshop, Paris, October 1999*.
- [5] Krishnamurti, Shriram, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *European Conference on Object-Oriented Programming*, 1998.
- [6] Leroy, Xavier et al. *The Objective Caml system*. Release 2.04, 1999.
- [7] Nordlander, Johan. Pragmatic subtyping in polymorphic languages. Proceedings of the *International Conference on Functional Programming*, 1998.
- [8] Nordlander, Johan. *Reactive Objects and Functional Programming*. PhD thesis, Chalmers University of Technology, Gteborg, 1999.
- [9] Oksanen, Kenneth. Real-time garbage collection of an almost unidirectional persistent heap. Submitted to ICFP2000.
- [10] Remy, Didier. Typing Record Concatenation for Free. *Nineteenth Annual Symposium on Principles Of Programming Languages*, pages 166–176, 1992.