Subtyping and Inheritance for Inductive Types

Erik Poll
University of Kent, Canterbury, UK

E.Poll@ukc.ac.uk

Abstract

Inheritance and subtyping are key features of object-oriented languages. We show that there are corresponding – or, more precisely, *dual* – notions for inductive types (or algebraic datatypes): there is a natural notion of subtyping for these types and an associated form of code reuse (inheritance) for programs on these types.

Inheritance and subtyping for inductive types not only suggest possible extensions of functional programming languages, but also provide a new perspective on inheritance as we know it from object-oriented languages, which may help to get a better understanding of this notion.

1 Introduction

Functional programming languages such as ML [MTH90] provide *algebraic datatypes* – e.g. lists, trees – , and type theories such as Coq [Cor95] or Alf [AGNvS94] provide a more general notion of *inductive type*. Algebras are a well-known way of modelling these types: algebraic datatypes and inductive types can be understood as term algebras (or initial algebras).

It has been observed that object types can be modelled as *co-algebras* [Rei95]. Co-algebras are not as well-known as algebras, but the importance of co-algebras has been recognised as a way of modelling not only objects, but also infinite data structures, coinductive types, and processes. (See [JR97] for a gentle introduction to co-algebras.) We don't have to know anything about co-algebras here, except that they are duals of algebras in some sense.

If algebraic datatypes can be modelled as algebras and object types as co-algebras, then algebraic data and objects can be seen as duals. This suggests that for constructions involving objects there may be corresponding – *dual* – constructions for algebraic datatypes. For objects we have subtyping and inheritance. Are there dual notions for algebraic datatypes? It turns out that there are: there is a natural notion of subtyping on algebraic types, and an associated form of code reuse for functions on algebraic types that is dual to inheritance. We will illustrate this in the setting of a functional programming language with algebraic datatypes. The dual of subtyping turns out to be *super*typing, which is related to subtyping in the obvious way: `A` is a subtype of `B` iff `B` is a supertype of `A`. The code reuse for functions on algebraic datatypes will be called *co-inheritance*, to distinguish it from the inheritance in object-oriented languages. Under the propositions-as-types isomorphism co-inheritance actually corresponds to a form of proof reuse for induction proofs that is commonly used.

We will only give an informal explanation of the duality with objects, just to show that what we describe really are duals of subtyping and inheritance in object-oriented languages. We will not give the definition of co-algebras here, nor will we describe how co-algebras can be used to model objects. And although the observation that objects and algebraic datatypes are duals comes from category theory, no category theory is used in this paper.

2 Algebraic Datatypes

In functional programming languages such as ML or Haskell we can define algebraic datatypes. For example, if `A` is some type, then a type of `A`-lists can be defined as

```
data ConsList = nil | cons A ConsList
```

Algebraic datatype are characterised by a set of *constructors* (`nil` and `cons` in the example above). These are the operations to construct elements of the algebraic datatype. The type `ConsList` can be understood as the smallest set containing `nil` and closed under `cons`-ing. Functions on algebraic types can be defined by pattern matching, for example

```
length : ConsList->Nat
length nil        = 0
length (cons a l) = 1 + length l
```

Remark 2.1 (Duality with OO) A small hint as to how this is dual to objects: constructors are the duals of *methods*. Just as an algebraic datatype is characterised by a set of constructors, object types are characterised by a set of methods. And whereas constructors are the only way to produce algebraic data, methods provide the only way to observe objects.

3 Subtyping

Consider a type of lists that not only provides an operation `cons` to add an element at the front of a list, but also provides an operation `snoc` to add an element at the end of a list:

```
data List = nil | cons A List | snoc List A
```

Note that `List` is just a term algebra, so for instance `(cons a nil)` and `(snoc nil a)` are different elements of `List`. It is possible to write functions on `List` that differentiate between `(cons a nil)` and `(snoc nil a)`, although we might want to avoid such functions.

The constructors of `ConsList` – `nil` and `cons` – are also constructors of `List`, and so every `ConsList` is also a `List`. This means that `ConsList` can be seen as a subtype of `List`, or, equivalently, `List` as a supertype of `ConsList`. We write this as `ConsList < List` .

The subtyping relation `<` comes with the following type inference rule for programs, known as the *subsumption* rule, which expresses the fact that subtypes are "subsets":

$$\frac{\mathtt{a : A} \qquad \mathtt{A < B}}{\mathtt{a : B}}$$

So if `l:ConsList`, then it follows from `ConsList < List` that `l:List`. The subtyping `ConsList < List` produces subtyping on more complicated types. For instance, for any type `B` we have

```
  B->ConsList < B->List ,
```

so if `f:B->ConsList` then also `f:B->List`. This means that we can immediately reuse all functions that produce elements of the old datatype `ConsList` as outputs to produce elements of the new datatype `List` as outputs.

To recap, we have shown that

$$\boxed{\text{adding constructors produces a supertype.}}$$

and that

$$\boxed{\begin{array}{c}\text{after adding constructors to produce a new (super)type,}\\ \text{programs that produce algebraic data as output can be reused.}\end{array}}$$

Remark 3.1 (Duality with OO) Recall that in object-oriented languages objects in a subclass can have more methods than objects in the superclass. For instance, to use the standard example, `ColouredPoint` could be a subclass of `Point`, i.e. `ColouredPoint < Point`, with instances of `ColouredPoint` having more methods than instances of `Point`.

So here the subtyping goes in the opposite direction as for algebraic types:

$$\boxed{\text{adding methods produces a } \textit{sub}\text{type.}}$$

Consequently, a different collection of functions can be reused. For example, a program that expects `Point`'s as *input* can be given `ColouredPoint`'s as inputs. This will not cause any problems, because any messages that can be sent to a `Point` can also be sent to a `ColouredPoint`. So

$$\boxed{\begin{array}{c}\text{after adding methods to produce a new (sub)type,}\\ \text{programs that take objects as } \textit{inputs} \text{ can immediately be reused.}\end{array}}$$

4 Co-inheritance

It does *not* follow from `ConsList < List` that `ConsList->B < List->B`. This is the (in)famous *contravariance* of `->` in its first argument. The subtyping rule for function types is

$$\frac{\mathtt{A}' < \mathtt{A} \qquad \mathtt{B} < \mathtt{B}'}{\mathtt{A\text{->}B} < \mathtt{A}'\mathtt{\text{->}B}'}$$

So we have `List->B < ConsList->B` and not `ConsList->B < List->B`. It makes sense that we do not have `ConsList->B < List->B`. Consider a typical function `f:ConsList->B` defined by pattern matching,

```
f : ConsList->B
f nil        = ...
f (cons a l) = ...
```

It is clear that applying this function `f` to a `List` may cause problems, because `f(snoc l a)` is not defined. (Of course we could allow `f` to be applied to `List`'s, and have it abort or diverge when it hits a `snoc`. But this rather defeats the purpose of typing our programs, which is the prevention of such run-time errors. Subtyping should only give "safe" inclusions between types, that will not introduce run-time errors.)

Since we do not have `ConsList->B < List->B`, we cannot reuse functions that accept `ConsList`'s as inputs and apply them to `List`'s. However, all is not lost. There is a natural way in which a function on `ConsList`'s such as `f` above can be reused to define a function on `List`'s. A typical definition for a function on `List`'s will be of the form

```
h : List -> B
h nil        = ...
h (cons a l) = ...
h (snoc l a) = ...
```

The only thing that is extra compared with the definition of `f` is the `snoc`-case. We could define `h` by *inheriting* the first two cases from `f`:

```
h : List -> B
co-inherits f : ConsList->B
h (snoc l a) = ...
```

This form of code reuse will be called *co-inheritance*. The definition of `h` above would be the same as the one obtained by copying the two defining clauses of `f` and replacing all occurrences of `f` by `h`.

For example, consider the following definition, which co-inherits the function `length` defined earlier in section 2:

```
newlength : List -> Nat
co-inherits length : ConsList->Nat
newlength (snoc l a) = 1 + newlength l
```

This definition is equivalent with

```
newlength : List -> Nat
newlength nil        = 0
newlength (cons a l) = 1 + newlength l
newlength (snoc l a) = 1 + newlength l
```

So the definition of `length (cons a l)`, "`1 + length l`", is copied as the definition of `newlength (cons a l)`, but instead of `length` now `newlength` is used to compute the recursive call.

An obvious thing to do is to give the function `newlength` the same name as `length`. This possibility is discussed in 4.1. And, as we discuss in 4.2, some restrictions have to be imposed on the function that is co-inherited if all definitions by co-inheritance are to be well-defined. But first we show that co-inheritance really is dual to inheritance in object-oriented languages.

The example above shows that

$$\boxed{\text{co-inheritance allows reuse of programs that take algebraic data as input.}}$$

Note that this nicely complements the reuse provide by the subtyping discussed in the section 3:

- Subtyping (`ConsList < List`) allows reuse of functions of type `B->ConsList` to produce `List`'s as *outputs*.
- Co-inheritance allows reuse of functions of type `ConsList->B` to accept `List`'s as *inputs*.

Note that these are different kinds of reusing code. The former is literally reusing the same code, the latter is reusing code in the sense of making incremental changes to existing code to produce new code. In 4.4 we show how co-inheritance also allows reuse of functions of type `ConsList->ConsList` to accept `List`'s as inputs and produce `List`'s as outputs.

Remark 4.1 (Duality with OO) Dualising the statement above predicts:

$$\boxed{\text{inheritance allows reuse of programs that produce objects as output.}}$$

We know that in object-oriented languages inheritance allows class definitions to be reused. A class definition does indeed provide a way to create objects, typically in the form of a function `new...` that produces objects as outputs. For example, the definition of a class `Point` could provide a function `newPoint:B->Point`, where the input of type `B` is used for initialisation. Think of `newPoint` as a function that takes some initial state of type `B` as input and wraps it up with a collection of methods (a method table) to produce an object. Inheritance would then allow us to reuse `newPoint:B->Point` when defining `newColouredPoint:B->ColouredPoint`.

So functions like `newPoint:B->Point` are the dual of functions like `f:ConsList->B` in the algebraic setting. Often a function such as `newPoint` will not take a argument, because there is some fixed initialisation, which obscures this duality somewhat.

As for algebras, subtyping and inheritance provide different kinds of reuse. Subtyping allows code to be reused without any change: client code for `Point`'s can be applied immediately be applied to `ColouredPoint`'s. Inheritance allows code to be reused in the sense of making an incremental change: `newColouredPoint` can be written by extending the definition `newPoint`.

4.1 Overloading

It would be nice to use the same name for `length:ConsList->Nat` and `newlength:List->Nat`, for instance calling them both `length`. This overloading would not cause any ambiguities; there would be two ways of interpreting `(length l)` for `l:ConsList`, namely

- as the original function `length:ConsList->B` applied to `l:ConsList`, or
- as the new function `length:List->B` applied to `l:List` (`ConsList < List`, so `l` also has type `List`).

However, it is clear that both interpretations give the same result. This absence of ambiguities in the presence of overloaded functions and subtyping is called *coherence*.

The overloading of `length` is a somewhat degenerated form of overloading, because it can be explained as just an instance of subtyping. The two types of the function `length` are `ConsList->Nat` and `List->Nat`. These are subtypes: `List->Nat < ConsList->Nat`. So we could just say that the type of `length` is `List->Nat`, since this automatically subsumes its other type `ConsList->Nat`. A more interesting example of overloading, which cannot be explained as just subtyping, is given in 4.4.

In the dual situation for objects, the idea of reusing the function name does *not* seem to make sense. We would not want to use the same name for `newPoint` and `newColouredPoint`. Still, one could imagine it would not do any harm to replace occurrences of `newPoint` by `newColouredPoint`.

4.2 Well-definedness

We have to impose a restriction on co-inheritance to ensure that definitions by co-inheritance are well-defined: the definition of `f:ConsList->B` that is co-inherited may not use other functions on `ConsList`. To understand why, consider a function

```
f : ConsList->B
f nil        = ...
f (cons a l) = ...(f' l)...
```

So `f` is defined in terms of another function `f':ConsList->B'`. If we were to define a function `h:List->B` by co-inheriting `f`,

```
h : List -> B
co-inherits f : ConsList->B
h (snoc l a) = ...
```

then applying `h` to a `List` might result in applying `f'` to a `List`, producing a type error.

We could however define `h:List->B` by co-inheriting `f` *after* defining `f':List->B` by co-inheriting `f'`, i.e. after extending the definition of `f'` to cope with `snoc`-lists. Note that for this it is crucial that the function that co-inherits `f'` is also called `f'`, otherwise the definition of `f` we inherit still refers to the old function `f'` that can only take `ConsList`'s as inputs.

4.3 Overriding

Instead of just adding clauses, as in the definition of `h` above, we could also *override* existing clauses. For instance, a function on `List` could redefine the value at `nil`:

```
length_plus_5 : List -> Nat
co-inherits length : ConsList->B
redefining length_plus_5 nil =  5
length_plus_5 (snoc l a) = 1 + length_plus_5 l
```

Clearly now the same name cannot be used for both the old and the new function, as this would introduce ambiguities.

Remark 4.2 (Duality with OO) Suppose that, in some object-oriented language with late binding, we define a class `Point` with a method `doublebump` that calls another method `bump`. At the time we write the definition of `doublebump` we do not know the code that will actually be executed for `bump`, because `bump` could be redefined in a subclass (e.g. `ColouredPoint`).

We now seem to have something similar for the definitions by pattern-matching. At the time we write the definition of `length (cons a l)` we do not know the code that will actually be executed to compute the recursive call on `l`. For instance, the original definition of `length (cons a l)`, "`1 + length l`", is still used to compute `length_plus_5 (cons a l)`, but now a different piece of code is executed to compute the recursive call on `l`, namely `length_plus_5`, which will produce a different result than `length` would. (The same thing already happens in the case of `newlength`, but there the new recursive call `newlength l` will produce the same result as the original call `length l` would, as least if `l` is a `ConsList`.)

To define the "new" value `length_plus_5 nil` we could use the "old" value `length nil`. For example,

```
length_plus_5 : List -> Nat
co-inherits length : ConsList->B
redefining length_plus_5 nil =  5 + length nil
```

Redefining of `length_plus_5` in terms of `length nil` looks like the dual of the use of "super", i.e. defining a "new" method of a subclass in terms of the "old" methods of the superclass.

4.4 "Real" overloading

In all examples we have seen so far co-inheriting a function of type `ConsList->B` produced a function of type `List->B`. This will not be the case if `ConsList` occurs in the output type `B`. For example, consider

```
tail : ConsList->ConsList
tail nil        = nil
tail (cons a l) = l
```

We could co-inherit `tail` to define a function on `List`'s, but the output of this new function will not be a `ConsList`, but a `List`.

```
tail : List->List
co-inherits tail : ConsList->ConsList
tail (snoc l a) = if (l = nil) then nil
                               else (snoc (tail l) a)
```

The overloading of the name `tail` can *not* be explained as subtyping, unlike the overloading of `length` discussed in 4.1. The function `tail` has types `ConsList->ConsList` and `List->List`. These two types are *not* in the subtype relation, and they do not even have a common subtype that could serve as the minimal type of `tail`. So the overloading of `tail` is "real" overloading, and not just subtyping. So co-inheritance provides a way to introduce "real" overloaded functions that are guaranteed to be coherent.

4.5 Co-inheritance is not supertyping

Co-inheritance may be possible even if there is no supertyping. Consider

```
data SnocList = nil | snoc A SnocList
```

Clearly `SnocList` is not a sub- or supertype of `ConsList`. Still, we could define a function `g:SnocList -> B` by co-inheriting the value at `nil` from a function `f:SnocList -> B`. For example,

```
snoclength : SnocList->Nat
co-inherits length : ConsList->Nat
snoclength (snoc l a) = 1 + snoclength l
```

There is actually a reason why we might want to define `snoclength` using co-inheritance rather than simply define `snoclength nil = 0`. We can give it the same name as `length`:

```
length : SnocList->Nat
co-inherits length : ConsList->Nat
length (snoc l a) = 1 + length l
```

So here co-inheritance is again used to overload a function name – `length` has type `ConsList->ConsList` and `SnocList->SnocList` – and again co-inheritance guarantees coherence.

5 Propositions as Types: Co-inheritance of Proofs

Co-inheritance is something most people will already have used, but for proofs rather than for programs! By the Curry-Howard isomorphism (propositions-as-types) constructing proofs by induction corresponds to defining functions by pattern-matching and (primitive) recursion, and so co-inheritance provides a way to reuse induction proofs. Co-inheritance of induction proofs is very common. Suppose we have given an proof by induction over A. If the (inductive) definition of A is later extended with another clause, then to update the proof we only have to add the corresponding new case in the induction proof.

For instance, suppose R be a relation defined by a set of rules (e.g. a reduction or typing relation) and suppose that we have proved $\forall x, y.\, xRy \Rightarrow P(x, y)$ by induction on the generation of xRy. If a new relation R' is defined by adding an extra rule to those for R, then to prove $\forall x, y.\, xR'y \Rightarrow P(x, y)$ we only have to prove the induction step for the extra rule. Of course this is only sound if no other properties of R are used, which is exactly the point made in 4.2 earlier. If other properties of R are used, then first we have to prove that these still hold for R' , which will again typically be done by just checking the extra case.

6 Inheritance vs Co-inheritance

We now give another example of subtyping and co-inheritance, which is more object-oriented in flavour. It shows that co-inheritance and inheritance can be used in similar situations, namely when new representations are added to a type (or, in OO terminology, a class).

Consider a datatype of `Shape`'s that can either be circles or squares, and a function `area` that computes the surface area of a shape:

```
data Shape = circle Point Num
           | square Point Num

area : Shape -> Num
area (circle centre radius)          = 0.5 * pi * square (radius)
area (square bottomleftcorner width) = square (width)
```

We can make a subtype `NewShape` of `Shape` by adding more constructors, and then define `area` for `NewShape`'s using co-inheritance. For example:

```
data NewShape = circle Point Num
              | square Point Num
              | rectangle Point Num Num

area : Shape -> Num
```

```
co-inherits area : Shape -> Num
area (rectangle bottomleftcorner width height) = width * height
```

Extending a type with a new representation (e.g. `Shape` with rectangles) is something that also happens in OO languages. We could have a *class* `Shape` with subclasses `Circle` and `Square`, and then decide to introduce a new subclass `Rectangle`. (We ignore the fact that `Square` should be a subclass of `Rectangle`.)

The difference with the OO approach is that in the example above no attempt is made to hide – or abstract away from – the representation of shapes. The constructors of `Shape` are visible for all to see, and functions on `Shape` can be defined by pattern matching in any part of the program. This is why when we add a new representation for rectangles we get a new type `NewShape` and cannot immediately reuse code written for `Shape`'s to deal with `NewShape`'s.

In the OO approach there would only be a select group of functions – the *methods* – that know about the representation of `Shape`'s, and the representation of `Shape`'s would be hidden from the rest of the program. Adding a new representation then only affects the methods, and we typically give new definitions of the methods for the new representation. All other code (the so-called client code) written for the old class can be reused without any change. Because of this, adding a new representation for rectangles to a class `Shape` does not have to produce a a new class `NewShape`, but we can still use the original class `Shape`.

7 Multiple Co-inheritance

It is straightforward to generalise the notion of co-inheritance to multiple co-inheritance. For example, recall the three algebraic types introduced earlier:

```
data ConsList = nil | cons A ConsList

data SnocList = nil | snoc A SnocList

data List = nil | cons A List | snoc List A
```

Clearly `List` is a supertype of both `SnocList` and `ConsList`: `ConsList < List` and `SnocList < List`. We can define a function on `List` by multiple co-inheritance, co-inheriting both a function on `SnocList` and a function on `ConsList`:

```
h : List -> B
co-inherits f: ConsList->B
        and g: SnocList->B
```

But both `ConsList` and `SnocList` have a constructor `nil`, so the value of `h` at `nil` could be co-inherited from `f` or `g`. So if `(f nil)` and `(g nil)` are not equal the definition above is ambiguous. Anyone familiar with object-oriented programming will notice that multiple inheritance can cause exactly the same problem! There are several ways to solve or avoid this problem:

- give priority to one of the functions that is co-inherited, e.g. the one mentioned first.
- forbid multiple co-inheritance in cases like this, i.e. where the subtypes have constructors in common.
- only allow multiple co-inheritance of `f` and `g` if the defining clauses of `f` and `g` for the shared constructors are inherited from a common source and hence identical. E.g. in the example above, if `g` inherits its value at `nil` from `f` or vice versa, or if `f` and `g` inherit their values at `nil` from some common "super" definition, then the definition of `h` would not be ambiguous.

Overloading

Like single co-inheritance, multiple co-inheritance can be used to introduce overloading: a function defined by co-inheritance can be given the same name as one of functions it co-inherits. If the domains of the functions it co-inherits have a constructor in common – e.g. `nil` in the example above – then we can only give it the name of the function that was given "priority". It is possible that the functions that we co-inheriting already have the same name. E.g. suppose `tail:SnocList->SnocList` is defined by inheriting `tail:ConsList->ConsList`:

```
 tail : SnocList->SnocList
 co-inherits tail : ConsList->ConsList
 tail (snoc l a) = if (l = nil) then nil
                               else (snoc (tail l) a)
```

Then `tail:List->List` can be defined as follows

```
tail : List->List
co-inherits tail : ConsList->ConsList
        and tail : SnocList->SnocList
```

Clearly it doesn't matter here from which function the `nil` case is co-inherited.

Well-definedness

Again we have to be careful with co-inheriting functions that rely on other functions. E.g. suppose `f:ConsList->B` and `g:SnocList->B` are defined using functions `f':ConsList->B` and `g':SnocList->B`:

```
 f : ConsList->B
 f nil        = ...
 f (cons a l) = (f' l)

 g : SnocList->B
 g nil        = ...
 g (snoc l a) = (g' l)
```

If we define `h:List->B` by co-inheriting `f` and `g`, then `f'` and `g'` – which expect `ConsList`'s and `SnocList`'s as arguments – may be invoked with `List`'s as arguments. As in the case of single co-inheritance, it would be possible to safely define `h:List->B` by co-inheriting `f` and `g` *after* upgrading the functions `f'` and `g'` to deal with `List`'s as arguments.

8 Conclusion

We have described notions of co-inheritance and subtyping for algebraic datatypes, that are duals of the inheritance and subtyping we know from OO. For algebraic datatypes

- adding *constructors* to an algebraic datatype produces a *super*type,
- subtyping allows reuse of programs that produce algebraic data as *output*,
- co-inheritance allows reuse of programs that take algebraic data as *input*.

For objects on the other hand,

- adding *methods* to a class produces a *sub*class,
- subtyping allows reuse of programs that accept objects an *input* (and send messages to them),

- inheritance allows reuse of programs that produce objects as *output* (i.e. class definitions).

Note that there are two different kinds of code reuse here. The code reuse made possible by subtyping is literally reuse of exactly the same code. The code reuse made possible by (co)inheritance is reuse in the sense of making incremental changes to existing code to produce new code.

There are several questions still unanswered. Co-inheritance and subtyping for algebraic types suggest possible extensions of functional programming languages. However, it is not clear how useful these would be, or what complications they would introduce. Also, what is the relation with other extensions of functional programming languages aimed at supporting code reuse or limited forms of object orientation? There are several of these extensions, for instance the class mechanism in Haskell [HHJW96] – which also allows some form of overloading –, the combination of this class mechanism with existential types [Läu96], and the experimental Haskell dialect called Mondrian [MC97].

The notion of co-inheritance for inductive types introduced here provides a different perspective on inheritance as we know it from object-oriented languages. This may help to get a better understanding of it. For example, for inductive types it is easier to see that subtyping and co-inheritance complement each other, in that they allow the reuse of different sets of functions.

One thing to be done is extending the description of objects as members of (terminal) co-algebras [Rei95] to account for inheritance and subtyping. This should make it easier to examine the relation between inheritance and co-inheritance. It remains to be seen if such an account of inheritance and subtyping for co-algebras would be a good description of these notions as they exist in real object-oriented languages.

References

[AGNvS94] Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström, and Björn von Sydow. A user's guide to ALF. Technical report, University of Göteborg/Chalmers, May 1994.

[Cor95] C. Cornes et al. The Coq proof assistant reference manual, version 5.10. Rapport technique RT-0177, INRIA, 1995.

[HHJW96] C. Hall, K. Hammond, S.L. Peyton Jones, and P. Wadler. Type classes in Haskell. *TOPLAS*, 18(2):pp.109–138, March 1996.

[JR97] Bart Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. In *EATCS Bulletin*. june 1997.

[Läu96] K. Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, May 1996.

[MC97] Erik Meijer and Koen Claessen. The design and implementation of Mondrian. In *Haskell Workshop*. ACM, June 1997.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[Rei95] Horst Reichel. An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science*, 5:129–152, 1995.