95

Baby Modula-3 and a theory of objects

Martín Abadi

February 2, 1993; revised December 2, 1992

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth. This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Robert W. Taylor, Director

To appear in Journal of Functional Programming, 2(4) 1994, Cambridge University Press.

Contents

Baby Modula-3 and a theory of objects

Martín Abadi

Abstract

Baby Modula-3 is a small, functional, object-oriented programming language. It is intended as a vehicle for explaining the core of Modula-3, from a biased perspective: Baby Modula-3 includes the main features of Modula-3 related to objects, but not much else. To the theoretician, Baby Modula-3 provides a tractable, concrete example of an object-oriented language, and we use it to study the formal semantics of objects.

Baby Modula-3 is defined with a structured operational semantics and with a set of static type rules. A denotational semantics guarantees the soundness of this definition.

1 Introduction

Baby Modula-3 is a small, functional, object-oriented programming language with a static type system. It is intended as a distillation and an explanation of the core of Modula-3 (Nelson, 1991), from a biased perspective: Baby Modula-3 includes the main features of Modula-3 related to objects, but not much else. To the theoretician, Baby Modula-3 provides a tractable, concrete example of an object-oriented language, and we use it to study the formal semantics of objects. Our study deals with both operational and denotational semantics.

The language is defined with a structured operational semantics in the style of Plotkin (1981) and with a set of type rules in the usual natural-deduction format. We prove a subject-reduction theorem, which implies that a well-typed program never produces run-time errors, such as the infamous "message not understood" error of Smalltalk.

Further, we give a denotational semantics for Baby Modula-3. The denotational semantics leads to a different proof that well-typed programs do not produce errors. It also provides a basis for axiomatic reasoning about Modula-3 programs, in that various rules for program verification can be proved sound with respect to the semantics.

The semantics of object types is based on a simple analogy with recursive record types. Intuitively, the type of objects with one field `f` of type `B` and one method `m` of return type `C` can be viewed as the type `T` of records with two fields, `f` of type `B` and `m` of type `T`$\rightarrow$`C`. The definition of `T` is recursive, since `T` appears in the type of `m`. This analogy has been suggested in the literature, in different frameworks; Cardelli (1986) and Mitchell (1990) have had some success with it. The analogy breaks, however, in the treatment of subtyping. The recursive record types used are not in the necessary subtype relations, because the recursion involved is not covariant. The difficulty in the combination of subtyping and recursion appears to be a well-known stumbling block in the folklore on object types. We surmount this difficulty with rather elementary techniques. The resulting semantics does not require much beyond recursive record types, but provides an adequate treatment of subtyping.

The next section is an informal overview. Section 3 describes the syntax of Baby Modula-3; Section 4 gives its semantics. The final section is a comparison with related works on the theory of typed object-oriented languages.

2 Overview

In this overview we introduce our treatment of objects, by comparing objects to records. We describe the expressions that denote objects in Baby Modula-3; we also describe the operations on objects, and present some typical reduction rules. Then we discuss issues in the design of type rules for Baby Modula-3. Finally, we present the main theme of our denotational semantics.

2.1 Expressions and their reduction

In Baby Modula-3, objects are made up of fields and methods; Baby Modula-3 is delegation-based in the sense that methods are directly attached to individual objects, and not to classes (see Section 5). We write `nil` for the object with no fields and no methods. If `a` is an object then we write $\mathtt{a}[\mathtt{f}=\mathtt{b},\mathtt{m}=\mathtt{c}]$ for `a`'s extension with a field `f` and a method `m`, with values `b` and `c`.

Throughout, we assume that the labels of fields and methods are taken from disjoint sets, so that it is always clear whether a field or a method is under consideration. Furthermore, we contemplate extension with exactly one field and one method at a time, for convenience and with no loss of generality. Although a field can be encoded as a method that ignores its argument, we treat fields explicitly, because the definitions and proofs for fields are good introductions to the corresponding ones for methods. In informal discussions, we sometimes lighten our notation, for example abbreviating $\mathtt{nil}[\mathtt{f}=\mathtt{b},\mathtt{m}=\mathtt{c}][\mathtt{f}'=\mathtt{b}',\mathtt{m}'=\mathtt{c}']$ by $[\mathtt{f}=\mathtt{b},\mathtt{m}=\mathtt{c},\mathtt{f}'=\mathtt{b}',\mathtt{m}'=\mathtt{c}']$.

An object is like an extensible record in many respects (e.g., (Wand, 1987; Cardelli, 1992)). For example, the object $[\mathtt{f}=\mathtt{b},\mathtt{m}=\mathtt{c}]$ is like the record with the fields `f` and `m` with the respective values `b` and `c`. The operations on objects correspond to operations on records, too:

- Objects can be extended with new fields and methods, much like records can be extended with new fields.
- New values can be assigned to the fields and methods of existing objects. For methods, this is called overriding, but in our treatment overriding is just assignment. (Modula-3 permits overriding only at the level of types; see Section 3.)
- Finally, fields can be read and methods can be invoked. Reading a field from an object is much like reading it from a record. On the other hand, invoking

a method has the effect of extracting its value, then applying this value to the object, and returning the result of the application.

We use the reduction rules that reflect the difference between fields and methods as an introduction to our structured operational semantics. In these rules, $\mathtt{a} \Rightarrow \mathtt{a}'$ may be read "the expression $\mathtt{a}$ reduces to the result $\mathtt{a}'$." Results are expressions of special forms; in particular, it is straightforward to determine whether a result represents an object and to examine its fields and methods. A result reduces only to itself.

$$\frac{\mathtt{a} \Rightarrow \mathtt{a}' \qquad \mathtt{f} = \mathtt{b} \ \mathtt{in} \ \mathtt{a}'}{\mathtt{a.f} \Rightarrow \mathtt{b}}$$

$$\frac{\mathtt{a} \Rightarrow \mathtt{a}' \qquad \mathtt{m} = \mathtt{c} \ \mathtt{in} \ \mathtt{a}' \qquad \mathtt{c}(\mathtt{a}') \Rightarrow \mathtt{d}}{\mathtt{a.m} \Rightarrow \mathtt{d}}$$

The first rule says: if $\mathtt{a}$ reduces to $\mathtt{a}'$, and $\mathtt{a}'$ is an object with the field $\mathtt{f}$ with the value $\mathtt{b}$, then $\mathtt{a.f}$ reduces to $\mathtt{b}$. On the other hand, the second rule says: if $\mathtt{a}$ reduces to $\mathtt{a}'$, and $\mathtt{a}'$ is an object with the method $\mathtt{m}$ with the value $\mathtt{c}$, and further the application $\mathtt{c}(\mathtt{a}')$ reduces to $\mathtt{d}$, then $\mathtt{a.m}$ reduces to $\mathtt{d}$.

The structured operational semantics includes many rules of the general form of the ones just given. There is one "normal" rule for each syntactic construction in the language. In addition, there are rules for reductions that produce a run-time error, represented by the special result `wrong`. For example, we have the rule:

$$\frac{\mathtt{a} \Rightarrow \mathtt{a}' \qquad \mathtt{no} \ \mathtt{f} \ \mathtt{in} \ \mathtt{a}'}{\mathtt{a.f} \Rightarrow \mathtt{wrong}}$$

It says: if $\mathtt{a}$ reduces to $\mathtt{a}'$, and $\mathtt{a}'$ is not an object with the field $\mathtt{f}$ (possibly not an object at all), then $\mathtt{a.f}$ reduces to `wrong`, that is, $\mathtt{a.f}$ produces a run-time error.

2.2 Types

In Baby Modula-3, certain types are distinguished as object types. There is a type of all objects, called `Root`. The type `Root` is the largest object type, and even the empty object `nil` has type `Root`. Object types can be extended, much like objects themselves: if `A` is an object type without the field `f` or the method `m` then $\mathtt{A}[\mathtt{f} : \mathtt{B}, \mathtt{m} : \mathtt{C}]$ is an object type, the extension of `A` with a field `f` of type `B` and a method `m` of return type `C`.

For example, $\mathtt{Root}[\mathtt{f} : \mathtt{Nat}, \mathtt{m} : \mathtt{Nat}][\mathtt{f}' : \mathtt{Nat}, \mathtt{m}' : \mathtt{Nat}]$ is an object type, which we may write $[\mathtt{f} : \mathtt{Nat}, \mathtt{m} : \mathtt{Nat}, \mathtt{f}' : \mathtt{Nat}, \mathtt{m}' : \mathtt{Nat}]$ adopting for object types an abbreviation analogous to the one for objects. Further, object $[\mathtt{f} = \mathtt{b}, \mathtt{m} = \mathtt{c}, \mathtt{f}' = \mathtt{b}', \mathtt{m}' = \mathtt{c}']$ has type $[\mathtt{f} : \mathtt{Nat}, \mathtt{m} : \mathtt{Nat}, \mathtt{f}' : \mathtt{Nat}, \mathtt{m}' : \mathtt{Nat}]$ if the values of the fields $\mathtt{f}$ and $\mathtt{f}'$ are always natural numbers and invoking the methods $\mathtt{m}$ and $\mathtt{m}'$ always returns natural numbers.

There are many ways of formalizing this informal description of object types and the example. Obtaining a sound, tractable, and useful set of type rules is not entirely straightforward.

According to our formulation, the object $[\mathtt{f} = \mathtt{b}, \mathtt{m} = \mathtt{c}, \mathtt{f}' = \mathtt{b}', \mathtt{m}' = \mathtt{c}']$ has type

$[\mathtt{f}:\mathtt{Nat},\mathtt{m}:\mathtt{Nat},\mathtt{f}':\mathtt{Nat},\mathtt{m}':\mathtt{Nat}]$ if $\mathtt{b}$ and $\mathtt{b}'$ have type $\mathtt{Nat}$ and $\mathtt{c}$ and $\mathtt{c}'$ have type $[\mathtt{f}:\mathtt{Nat},\mathtt{m}:\mathtt{Nat},\mathtt{f}':\mathtt{Nat},\mathtt{m}':\mathtt{Nat}]\rightarrow\mathtt{Nat}$. Since we preserve membership in the type $[\mathtt{f}:\mathtt{Nat},\mathtt{m}:\mathtt{Nat},\mathtt{f}':\mathtt{Nat},\mathtt{m}':\mathtt{Nat}]$ as an invariant for the object, the methods $\mathtt{m}$ and $\mathtt{m}'$ are invoked only with arguments of this type. This motivates the condition that $\mathtt{c}$ and $\mathtt{c}'$ have type $[\mathtt{f}:\mathtt{Nat},\mathtt{m}:\mathtt{Nat},\mathtt{f}':\mathtt{Nat},\mathtt{m}':\mathtt{Nat}]\rightarrow\mathtt{Nat}$.

Generalizing from this example, we obtain the following type rule for overriding (a slightly simplified, weakened version of the rule presented in Section 3):

$$\frac{\mathtt{E} \vdash \mathtt{a}:\mathtt{A} \qquad \mathtt{E} \vdash \mathtt{c}:\mathtt{A}\rightarrow\mathtt{C} \qquad \mathtt{m}:\mathtt{C}\ \mathtt{in}\ \mathtt{A}}{\mathtt{E} \vdash \mathtt{a.m} := \mathtt{c}:\mathtt{A}}$$

This rule may be read: given the environment $\mathtt{E}$, if $\mathtt{a}$ has type $\mathtt{A}$, $\mathtt{c}$ is a function from $\mathtt{A}$ to $\mathtt{C}$, and $\mathtt{A}$ is an object type with method $\mathtt{m}$ with return type $\mathtt{C}$, then it is legal to assign $\mathtt{c}$ to $\mathtt{a.m}$, and the new value of $\mathtt{a}$ has type $\mathtt{A}$. A similar rule deals with adding a method to an object. As explained below, both overriding and extension are functional operations in Baby Modula-3, but the corresponding type rules would be sensible for an imperative language as well.

The type system also includes a subtype relation $\leq$. If $\mathtt{A} \leq \mathtt{B}$ and $\mathtt{a}$ has type $\mathtt{A}$ then $\mathtt{a}$ has type $\mathtt{B}$. The central rule for subtyping says that if $\mathtt{B}$ is an object type and $\mathtt{A}$ an extension of $\mathtt{B}$ then $\mathtt{A} \leq \mathtt{B}$, so for example $[\mathtt{f}:\mathtt{Nat},\mathtt{m}:\mathtt{Nat}] \leq \mathtt{Root}$ and $[\mathtt{f}:\mathtt{Nat},\mathtt{m}:\mathtt{Nat},\mathtt{f}':\mathtt{Nat},\mathtt{m}':\mathtt{Nat}] \leq [\mathtt{f}:\mathtt{Nat},\mathtt{m}:\mathtt{Nat}]$. Modula-3 allows single inheritance, and not multiple inheritance; the order of fields and methods matters in determining whether two object types are in the subtype relation. For example, the two types $[\mathtt{f}:\mathtt{Nat},\mathtt{m}:\mathtt{Nat},\mathtt{f}':\mathtt{Nat},\mathtt{m}':\mathtt{Nat}]$ and $[\mathtt{f}':\mathtt{Nat},\mathtt{m}':\mathtt{Nat},\mathtt{f}:\mathtt{Nat},\mathtt{m}:\mathtt{Nat}]$ are incomparable. Our rules correspond to single inheritance, but there would be little difficulty in dealing with multiple inheritance instead. Our semantics already models multiple inheritance.

Finally, the type system includes a recursive-type construction. Recursive types arise often in dealing with objects, and for example the type of all objects that contain a field $\mathtt{f}$ of type $\mathtt{Nat}$ and a binary method $\mathtt{m}$ of return type $\mathtt{Nat}$ is the solution to the equation:

$$\mathtt{X} = [\mathtt{f}:\mathtt{Nat},\mathtt{m}:\mathtt{X}\rightarrow\mathtt{Nat}]$$

There has been interesting recent literature on the interaction between subtyping and recursive types (e.g., (Amadio and Cardelli, 1991)). In our language this interaction remains simple, and in particular it does not give rise to any special rules.

The result that connects reduction and typing is the subject-reduction theorem. It says that types are not lost by reduction: if $\mathtt{a} \Rightarrow \mathtt{a}'$ and $\vdash \mathtt{a}:\mathtt{A}$ then $\vdash \mathtt{a}':\mathtt{A}$. It is a simple corollary that if $\vdash \mathtt{a}:\mathtt{A}$ then $\mathtt{a}$ does not reduce to $\mathtt{wrong}$, and thus its evaluation does not produce a run-time error.

2.3 Denotational semantics

The denotational semantics of Baby Modula-3 needs to address the issues that arise from the typing of methods and from the use of subtyping and recursive types. As indicated, the main theme of our interpretation is the analogy between object types and recursive record types.

The type rules for Baby Modula-3 suggest comparing an object type such as $[\mathtt{f}:\mathtt{Nat},\mathtt{m}:\mathtt{Nat},\mathtt{f}':\mathtt{Nat},\mathtt{m}':\mathtt{Nat}]$ to a recursive record type, here the type T of all records with fields f and $\mathtt{f}'$ with type Nat and with fields m and $\mathtt{m}'$ with type $\mathtt{T}\to\mathtt{Nat}$. We may define T as the solution to the equation

$$\mathtt{X} = \langle\!\langle \mathtt{f}:\mathtt{Nat},\mathtt{m}:\mathtt{X}\to\mathtt{Nat},\mathtt{f}':\mathtt{Nat},\mathtt{m}':\mathtt{X}\to\mathtt{Nat}\rangle\!\rangle$$

where $\langle\!\langle \mathtt{fm}:\mathtt{Am},\ldots,\mathtt{fn}:\mathtt{An}\rangle\!\rangle$ denotes the type of records with fields fm, ..., fn with respective types Am, ..., An. Perhaps because of the simplicity of Baby Modula-3 (and of Modula-3) this analogy gets us quite far. To support it, we adapt standard methods for the solution of recursive type equations, with only a few technical surprises.

But the analogy stops working when we consider the rules for subtyping. In our example, the problem is that the type function

$$F(\mathtt{X}) = \langle\!\langle \mathtt{f}:\mathtt{Nat},\mathtt{m}:\mathtt{X}\to\mathtt{Nat},\mathtt{f}':\mathtt{Nat},\mathtt{m}':\mathtt{X}\to\mathtt{Nat}\rangle\!\rangle$$

is contravariant in X and we take its fixpoint in constructing a recursive record type. The result of taking a fixpoint of a contravariant function is unpredictable in general. In particular, even if F is pointwise smaller than G (say, in the $\leq$ relation), it does not follow that the fixpoint of F is smaller than the fixpoint of G when F and G are allowed to be arbitrary contravariant functions. In our example, again, a bit of care in the definitions yields that $F(\mathtt{X})$ is a subtype of

$$G(\mathtt{X}) = \langle\!\langle \mathtt{f}:\mathtt{Nat},\mathtt{m}:\mathtt{X}\to\mathtt{Nat}\rangle\!\rangle$$

for each X, but the fixpoints of the two functions are unrelated. Hence the simple interpretation of object types based on recursive record types does not validate

$$[\mathtt{f}:\mathtt{Nat},\mathtt{m}:\mathtt{Nat},\mathtt{f}':\mathtt{Nat},\mathtt{m}':\mathtt{Nat}] \;\leq\; [\mathtt{f}:\mathtt{Nat},\mathtt{m}:\mathtt{Nat}]$$

The solution to this problem remains elementary. The meaning of an object type A is defined to include all the values allowed by the meaning of A as a recursive record type, and also all the values allowed by the meaning of any extension of A as a recursive record type. It seems somewhat remarkable that such a simple solution does not introduce any new problems. Its only apparent disadvantage is that it may be hard to formulate the modified interpretation of object types within a usual typed language such as System F (Girard, 1972); and perhaps this is why it was not previously noticed. The modified definition can be given in our semantic framework. The resulting denotational semantics is sound, in that it validates both the reduction rules and the type rules.

3 Syntax

In this section we discuss syntactic aspects of Baby Modula-3. First we present all relevant definitions, for reduction and typing, and then we start the syntactic study of the language.

3.1 Expressions

The grammar for terms is:

$$
\begin{array}{rcl}
\mathtt{a} & ::= & \mathtt{x} \\
& | & \mathtt{fun(x : A)b} \\
& | & \mathtt{b(a)} \\
& | & \mathtt{nil} \\
& | & \mathtt{a[f = b, m = c]} \\
& | & \mathtt{a.f} \\
& | & \mathtt{a.m} \\
& | & \mathtt{a.f := b} \\
& | & \mathtt{a.m := c} \\
& | & \mathtt{wrong}
\end{array}
$$

First we have variables, abstraction, and application. In the abstraction appears an expression `A`. It is intended that `A` range over type expressions, but the grammar for terms does not make a commitment to a particular type structure. Many type systems are possible; in the extreme type system with only one type expression, we have an untyped calculus. Subsection 3.2 includes a particular definition for type expressions. The operational and denotational semantics of terms do not depend on this definition.

The object constructs are `nil` (the object with no fields or methods), object extension ($\mathtt{a[f = b, m = c]}$), field reading ($\mathtt{a.f}$), method invocation ($\mathtt{a.m}$), assignment to a field ($\mathtt{a.f := b}$), and method overriding ($\mathtt{a.m := c}$). Note that the assignment expressions are terms, not commands; an assignment for an object `a` is intended to return the resulting value of `a`. Finally, we have `wrong`, the representation of a run-time error.

Relation to Modula-3

As a further explanation of the grammar just given, we briefly compare Baby Modula-3 to Modula-3. Readers not familiar with Modula-3 may want to skip this comparison.

The syntax for variables and for applications in Modula-3 is the same as here; abstractions are given with an explicit name `N` and an explicit return type `B`, in the form $\mathtt{PROCEDURE\ N(x : A) :\ B\ =\ RETURN\ b\ N}$; the constant `nil` is commonly written `NIL`.

In Modula-3, objects are not built by extension. Rather, they are allocated completely at once, with calls to `NEW`. In order to create the object that we write $\mathtt{nil[fd = bd, me = ce]\ldots[fi = bi, mj = cj]}$, the call is

$$\mathtt{NEW(A, fd := bd, me := ce, \ldots, fi := bi, mj := cj)}$$

with `A` the type of the object created. The methods `ce`, ..., `cj` could be arbitrary expressions in the original Modula-3 (Cardelli *et al.*, 1988). They must be top-level procedure constants in the current Modula-3.

The Modula-3 syntax for reading a field and for assignment to a field is the same as here; the method invocation `a.m` is written `a.m()`. Modula-3 does not allow overriding of methods at the value level, but only at the type level. (Type declarations may include values for fields and methods, which are used as defaults in calls to `NEW`. Overriding at the type level means declaring a type with a new default.) Overriding at the value level does appear in other languages (e.g., (Steele, 1990)) and it is noticeably absent from the class-based languages discussed in Section 5. We include it for completeness and because its formal treatment remains simple, perhaps surprisingly so.

3.2 Type rules

Subsections 3.2.1 to 3.2.5 introduce the type rules of Baby Modula-3. The experienced reader may wish to skim the first three of these subsections to focus on the last two, which contain the rules for subtyping and those for relating values and types.

3.2.1 Environments

We start with three rules for proving judgements of the form $\vdash \mathtt{E}$, read "`E` is a legal environment." An environment is a list; `empty` denotes the empty list and a comma denotes list concatenation.

$$\frac{}{\vdash \mathtt{empty}}$$

$$\frac{\vdash \mathtt{E} \qquad \mathtt{X\ not\ in\ E}}{\vdash \mathtt{E, X}}$$

$$\frac{\mathtt{E} \vdash \mathtt{A} \qquad \mathtt{x\ not\ in\ E}}{\vdash \mathtt{E, x : A}}$$

3.2.2 Types

Next come some rules for proving judgements of the form $\mathtt{E} \vdash \mathtt{A}$, read "`A` is a legal type in environment `E`," and of the form $\mathtt{E} \vdash \mathtt{A\ obj}$, read "`A` is a legal object type in environment `E`." An expression `A` such that $\mathtt{E} \vdash \mathtt{A}$ for some `E` is called a type expression. An expression `A` such that $\mathtt{E} \vdash \mathtt{A\ obj}$ for some `E` is called an object type expression.

$$\frac{\vdash \mathtt{E1, X, E2}}{\mathtt{E1, X, E2} \vdash \mathtt{X}}$$

$$\frac{\mathtt{E} \vdash \mathtt{A\ obj}}{\mathtt{E} \vdash \mathtt{A}}$$

$$\frac{\mathtt{E} \vdash \mathtt{A} \qquad \mathtt{E} \vdash \mathtt{B}}{\mathtt{E} \vdash \mathtt{A} \to \mathtt{B}}$$

$$\frac{\vdash \mathtt{E}}{\mathtt{E} \vdash \mathtt{Root\ obj}}$$

$$\frac{\mathtt{E} \vdash \mathtt{A\ obj} \qquad \mathtt{E} \vdash \mathtt{B} \qquad \mathtt{E} \vdash \mathtt{C} \qquad \mathtt{no\ f,m\ in\ A}}{\mathtt{E} \vdash \mathtt{A[f:B,m:C]\ obj}}$$

$$\frac{\mathtt{E,X} \vdash \mathtt{A\ obj}}{\mathtt{E} \vdash \mathtt{Mu(X)A\ obj}}$$

Here `no f in A` means that `f : B in A` holds for no `B`, and `f : B in A` is defined by induction on `A` to mean that `A` advertises the field `f` with type `B`:

- `f : B in A[f : B, m : C]`;
- if `f : B in A` then `f : B in A[f' : B', m' : C']`;
- if `f : B in A` then `f : B[Mu(X)A/X] in Mu(X)A`.

The definition for methods is analogous. The relations `f : B in A` and `m : C in A` are decidable, as they can easily be computed following their inductive definitions.

In examples, we use the types `Nat` and `Real`, but we do not treat them formally.

Discussion

The expression `Mu(X)A` represents a recursive type `B` such that `B = A[B/X]`. Note that the only recursive types allowed are object types. This restriction is easy to formulate using the judgments that distinguish object type expressions, `E ⊢ A obj`. There is neither difficulty nor fundamental gain in removing this restriction.

Note also that environments may include the assumption that `X` is a type, but not that it is an object type. This means that object type expressions can be built by extension and recursion from `Root` but not from type variables. For example, `X[f : Nat, m : Nat]` is not an object type expression, and in fact it is not a type expression at all. In further work (with Cardelli), we hope to be able to treat object type variables by using kinds to classify types.

Relation to Modula-3

The Modula-3 syntax for `A→B` is `PROCEDURE(x : A) : B`; the formal parameter `x` is made explicit. The type `Root` is commonly written `ROOT`; and `A[f : B, m : C]` is written `A OBJECT f : B METHODS m() : C END`. In Modula-3, recursive types are not expressed with the `Mu` construct and with type variables; rather, they are declared with equations such as `TYPE A = OBJECT f : A END`.

In Modula-3, the type `NULL`, which contains only `NIL`, is a subtype of every object type. We do not have an analogue of `NULL`, and in fact `nil` is not in every object type.

3.2.3 Type equalities

The rules for type equality deal with judgements of the form $\mathtt{E} \vdash \mathtt{A} = \mathtt{B}$, read "$\mathtt{A}$ and $\mathtt{B}$ are equal types in environment $\mathtt{E}$." Type equality is defined to be a congruence on type expressions. In addition, equality rules for recursive types have the effect of equating two type expressions whenever the infinite trees obtained from them by unfolding are equal.

We omit the rules for type equality; the interested reader can consult the work of Amadio and Cardelli (1991). In what follows, we sometimes identify types that are provably equal, for simplicity.

3.2.4 Subtypes

The rules for subtyping deal with judgements of the form $\mathtt{E} \vdash \mathtt{A}{\leq}\mathtt{B}$, read "$\mathtt{A}$ is a subtype of $\mathtt{B}$ in environment $\mathtt{E}$." Subtyping is reflexive and transitive. The only nontrivial subtyping is that between an extension of an object type and the object type, so that in particular `Root` is the largest object type. The function-space constructor $\rightarrow$ is neither covariant nor contravariant. Moreover, inheritance is simple and not multiple.

$$\frac{\mathtt{E} \vdash \mathtt{A} = \mathtt{B}}{\mathtt{E} \vdash \mathtt{A}{\leq}\mathtt{B}}$$

$$\frac{\mathtt{E} \vdash \mathtt{A}{\leq}\mathtt{B} \qquad \mathtt{E} \vdash \mathtt{B}{\leq}\mathtt{C}}{\mathtt{E} \vdash \mathtt{A}{\leq}\mathtt{C}}$$

$$\frac{\mathtt{E} \vdash \mathtt{A\ obj} \qquad \mathtt{E} \vdash \mathtt{B} \qquad \mathtt{E} \vdash \mathtt{C} \qquad \mathtt{no\ f,m\ in\ A}}{\mathtt{E} \vdash \mathtt{A[f:B,m:C]}{\leq}\mathtt{A}}$$

Discussion

To illustrate the use of the subtyping rules, we can show that $\mathtt{B}{\leq}\mathtt{A}$, where $\mathtt{A}$ and $\mathtt{B}$ are defined recursively by

$$\begin{aligned} \mathtt{A} &= \mathtt{Mu(X)Root[f:X,m:X]} \\ \mathtt{B} &= \mathtt{Mu(Y)Root[f:A,m:A][f':A',m':A']} \end{aligned}$$

and $\mathtt{A'}$ is an arbitrary type expression. The subtyping proof starts by unfolding once each of $\mathtt{A}$ and $\mathtt{B}$:

$$\begin{aligned} \mathtt{A} &= \mathtt{Root[f:A,m:A]} \\ \mathtt{B} &= \mathtt{Root[f:A,m:A][f':A'[B/Y],m':A'[B/Y]]} \end{aligned}$$

Then the rule for subtyping object types is applicable, and yields the desired result. This proof does not rely on any special rules for subtyping recursive types. However, in a more general context, rules for subtyping recursive types would be wanted (as in (Amadio and Cardelli, 1991)).

In contrast, note that it is not provable that $\mathtt{B}{\leq}\mathtt{A}$, where $\mathtt{A}$ and $\mathtt{B}$ are defined recursively by

$$\begin{aligned} \mathtt{A} &= \mathtt{Mu(X)Root[f : X, m : X]} \\ \mathtt{B} &= \mathtt{Mu(Y)Root[f : Y, m : Y][f' : A', m' : A']} \end{aligned}$$

This subtyping would hold under an additional hypothesis: that the extension type constructor is monotonic with respect to the subtype relation (so that if $\mathtt{Z} \leq \mathtt{Z}'$ and $\mathtt{W} \leq \mathtt{W}'$ then $\mathtt{Root[f : Z, m : W]} \leq \mathtt{Root[f : Z', m : W']}$). It would then be provable that $\mathtt{B} \leq \mathtt{A}$, using sound rules such as those of Amadio and Cardelli.

However, the additional monotonicity hypothesis is unsound in general. It is not hard to construct examples that illustrate this unsoundness. Consider the types $\mathtt{C} = \mathtt{Root[f : Nat, m : Nat]}$ and $\mathtt{D} = \mathtt{Root[f : Real, m : Nat]}$, with $\mathtt{Nat} \leq \mathtt{Real}$. We can build an object `a` of type `C` where the method `m` returns the value of the field `f`: let $\mathtt{a} = \mathtt{nil[f = 0, m = fun(z : C)(z.f)]}$. Suppose that $\mathtt{C} \leq \mathtt{D}$, and thus that `a` has type `D`. Then we can write $(\mathtt{a.f} := \pi)\mathtt{.m}$ and expect to obtain a result of type `Nat`, rather than π; hence, $\mathtt{C} \not\leq \mathtt{D}$.

Similar examples demonstrate that extending Baby Modula-3 with subtypings such as $\mathtt{B} \leq \mathtt{A}$ would be unsound as well. In some programs, the absence of these subtypings can be an obstacle. A simple remedy consists in incorporating dynamic typing into the language, as in Modula-3. With Cardelli, we are investigating a more complex but more ambitious remedy based on polymorphism

3.2.5 Typechecking

The typechecking rules are based on judgements of the form $\mathtt{E} \vdash \mathtt{a : A}$, read "`a` has type `A` in environment `E`." In the rules for typechecking objects, we use auxiliary judgements of the form $\mathtt{E} \vdash \mathtt{a : A \;\; Self{=}S}$. These are four-place judgements, relating an environment `E`, a term `a`, and two types `A` and `S`; `Self=` is simply a keyword. Next we list the rules; the first one is called the subsumption rule.

$$\frac{\mathtt{E} \vdash \mathtt{b : A} \qquad \mathtt{E} \vdash \mathtt{A \leq B}}{\mathtt{E} \vdash \mathtt{b : B}}$$

$$\frac{\vdash \mathtt{E1, x : A, E2}}{\mathtt{E1, x : A, E2} \vdash \mathtt{x : A}}$$

$$\frac{\mathtt{E, x : A} \vdash \mathtt{b : B}}{\mathtt{E} \vdash \mathtt{fun(x : A)b : A {\rightarrow} B}}$$

$$\frac{\mathtt{E} \vdash \mathtt{c : A {\rightarrow} B} \qquad \mathtt{E} \vdash \mathtt{a : A}}{\mathtt{E} \vdash \mathtt{c(a) : B}}$$

$$\frac{\mathtt{E} \vdash \mathtt{a : A \;\; Self{=}A}}{\mathtt{E} \vdash \mathtt{a : A}}$$

$$\frac{\mathtt{E} \vdash \mathtt{S}}{\mathtt{E} \vdash \mathtt{nil : Root \;\; Self{=}S}}$$

$$\frac{\texttt{E} \vdash \texttt{a}:\texttt{A}\ \ \texttt{Self=S} \qquad \texttt{E} \vdash \texttt{b}:\texttt{B} \qquad \texttt{E} \vdash \texttt{c}:\texttt{D}{\rightarrow}\texttt{C} \quad \texttt{E} \vdash \texttt{S}{\leq}\texttt{D} \qquad \texttt{no f,m in A}}{\texttt{E} \vdash \texttt{a}[\texttt{f}=\texttt{b},\texttt{m}=\texttt{c}]:\texttt{A}[\texttt{f}:\texttt{B},\texttt{m}:\texttt{C}]\ \ \texttt{Self=S}}$$

$$\frac{\texttt{E} \vdash \texttt{a}:\texttt{A} \qquad \texttt{E} \vdash \texttt{A obj} \qquad \texttt{f}:\texttt{B in A}}{\texttt{E} \vdash \texttt{a.f}:\texttt{B}}$$

$$\frac{\texttt{E} \vdash \texttt{a}:\texttt{A} \qquad \texttt{E} \vdash \texttt{A obj} \qquad \texttt{m}:\texttt{C in A}}{\texttt{E} \vdash \texttt{a.m}:\texttt{C}}$$

$$\frac{\texttt{E} \vdash \texttt{a}:\texttt{A} \qquad \texttt{E} \vdash \texttt{b}:\texttt{B} \qquad \texttt{f}:\texttt{B in A}}{\texttt{E} \vdash \texttt{a.f} := \texttt{b}:\texttt{A}}$$

$$\frac{\texttt{E} \vdash \texttt{a}:\texttt{A} \qquad \texttt{E} \vdash \texttt{c}:\texttt{D}{\rightarrow}\texttt{C} \qquad \texttt{E} \vdash \texttt{A}{\leq}\texttt{D} \qquad \texttt{m}:\texttt{C in A}}{\texttt{E} \vdash \texttt{a.m} := \texttt{c}:\texttt{A}}$$

Discussion

The use of auxiliary judgements deserves explanation. When `S` is an object type, the proof of $\texttt{E} \vdash \texttt{s}:\texttt{S}$ is reduced to the proof of $\texttt{E} \vdash \texttt{s}:\texttt{S}\ \ \texttt{Self=S}$, and later to similar proofs $\texttt{E} \vdash \texttt{a}:\texttt{A}\ \ \texttt{Self=S}$, where `s` and `S` are extensions of `a` and `A`, respectively. The argument `S` preserves a record of what the original typechecking problem was. This is needed for typechecking methods in `a`; intuitively, `S` is taken as the type of "self" (`s`), the argument of the methods. In the rule for object extensions, methods are required to map `S`, or some supertype `D` of `S`, to the appropriate return type. The auxiliary type `D` is introduced for generality, to compensate for the omission of a rule of contravariance of $\rightarrow$ in its first argument. With this rule, it would be equivalent to use `S` instead of `D`. A similar use of `D` is made in the rule for overriding.

As an example, consider the problem of deriving the judgment $\vdash \texttt{s}:\texttt{S}$ when `S` is $\texttt{Root}[\texttt{f}:\texttt{Nat},\texttt{m}:\texttt{Nat}][\texttt{f}':\texttt{Nat},\texttt{m}':\texttt{Nat}]$ and `s` is $\texttt{nil}[\texttt{f}=\texttt{b},\texttt{m}=\texttt{c}][\texttt{f}'=\texttt{b}',\texttt{m}'=\texttt{c}']$. Using the rules given, it suffices to prove the judgment $\vdash \texttt{s}:\texttt{S}\ \ \texttt{Self=S}$. In turn, $\vdash \texttt{s}:\texttt{S}\ \ \texttt{Self=S}$ can be obtained from the judgments $\vdash \texttt{b}':\texttt{Nat}$, $\vdash \texttt{c}':\texttt{S}{\rightarrow}\texttt{Nat}$, and $\vdash \texttt{nil}[\texttt{f}=\texttt{b},\texttt{m}=\texttt{c}]:\texttt{Root}[\texttt{f}:\texttt{Nat},\texttt{m}:\texttt{Nat}]\ \ \texttt{Self=S}$; this last judgment can itself be proved from $\vdash \texttt{b}:\texttt{Nat}$ and $\vdash \texttt{c}:\texttt{S}{\rightarrow}\texttt{Nat}$. Note that the method `m` in `s` will always be applied to an element of `S`, and that the condition $\vdash \texttt{c}:\texttt{S}{\rightarrow}\texttt{Nat}$ allows `c` to be $\texttt{fun}(\texttt{x}:\texttt{S})(\texttt{x.m}')$.

We deliberately omit any mechanism for unfolding recursive types in judgements of the form $\texttt{E} \vdash \texttt{a}:\texttt{A}\ \ \texttt{Self=S}$. If `A` is a recursive type, then $\texttt{E} \vdash \texttt{a}:\texttt{A}\ \ \texttt{Self=S}$ is never provable. While the omission makes the type system simpler, it does not result in a loss of power. For example, if we were to include that mechanism, the proof of $\vdash \texttt{nil}:\texttt{Mu(X)Root}$ could be reduced to that of $\vdash \texttt{nil}:\texttt{Mu(X)Root}\ \ \texttt{Self=Mu(X)Root}$; but it can also be reduced by unfolding to the proof of $\vdash \texttt{nil}:\texttt{Root}$, and this proof succeeds without any new mechanism for unfolding.

On the whole, the type rules are a little restrictive. In particular, they mean that extensions may be applied only to `nil`, to construct objects of the form $\texttt{nil}[\texttt{fd}=\texttt{bd},\texttt{me}=\texttt{ce}]\ldots[\texttt{fi}=\texttt{bi},\texttt{mj}=\texttt{cj}]$. (Thus, the use of a general extension syntax for objects is mostly a matter of taste.) For example, the rules do not provide a type for the function $\texttt{fun}(\texttt{x} : \texttt{Root}[\texttt{f} : \texttt{Nat}, \texttt{m} : \texttt{Nat}])(\texttt{x}[\texttt{f}' = \texttt{0}, \texttt{m}' = \texttt{fun}(\texttt{y} : \texttt{Root})\texttt{0}])$ where extension is applied to a variable in the body. This limitation would disappear were we to include suitable subsumption rules among the type rules with **Self=**. These new rules seem sound, and they may be of some interest.

3.3 Reduction rules

In the structured operational semantics, some closed expressions are viewed as proper results: the function results $\texttt{fun}(\texttt{x} : \texttt{A})\texttt{b}$, and the object results `nil` and

$$\texttt{nil}[\texttt{fd} = \texttt{bd}, \texttt{me} = \texttt{ce}] \ldots [\texttt{fi} = \texttt{bi}, \texttt{mj} = \texttt{cj}]$$

where all of `bd`, `ce`, ..., `bi`, `cj` are proper results and all the labels are distinct. All proper results are results, and in addition `wrong` is a result.

We write $\texttt{f} = \texttt{b}\ \texttt{in}\ \texttt{a}$ when `a` is an object result of the form

$$\texttt{nil} \ldots [\texttt{f} = \texttt{b}, \ldots] \ldots [\texttt{fi} = \texttt{bi}, \texttt{mj} = \texttt{cj}]$$

then write $\texttt{a}\{\texttt{f} \leftarrow \texttt{b}'\}$ for the result of replacing `b` with $\texttt{b}'$, obtaining

$$\texttt{nil} \ldots [\texttt{f} = \texttt{b}', \ldots] \ldots [\texttt{fi} = \texttt{bi}, \texttt{mj} = \texttt{cj}]$$

and write $\texttt{f}\ \texttt{in}\ \texttt{a}$ when `a` is an object result and $\texttt{f} = \texttt{b}\ \texttt{in}\ \texttt{a}$ holds for some `b`. The notations $\texttt{m} = \texttt{c}\ \texttt{in}\ \texttt{a}$, $\texttt{a}\{\texttt{m} \leftarrow \texttt{c}'\}$, and $\texttt{m}\ \texttt{in}\ \texttt{a}$ have analogous definitions.

The reduction relation $\texttt{a} \Rightarrow \texttt{b}$ ("`a` reduces to `b`") is axiomatized by the rules below. It is a binary relation between closed expressions and results. It is easy to see that each expression reduces to at most one result, and that a result reduces only to itself.

$$\frac{}{\texttt{fun}(\texttt{x} : \texttt{A})\texttt{b} \Rightarrow \texttt{fun}(\texttt{x} : \texttt{A})\texttt{b}}$$

$$\frac{\texttt{a} \Rightarrow \texttt{a}'\ (\neq \texttt{wrong}) \qquad \texttt{b} \Rightarrow \texttt{fun}(\texttt{x} : \texttt{A})\texttt{b}' \qquad \texttt{b}'[\texttt{a}'/\texttt{x}] \Rightarrow \texttt{b}''}{\texttt{b}(\texttt{a}) \Rightarrow \texttt{b}''}$$

$$\frac{\texttt{a} \Rightarrow \texttt{wrong}}{\texttt{b}(\texttt{a}) \Rightarrow \texttt{wrong}}$$

$$\frac{\texttt{a} \Rightarrow \texttt{a}' \qquad \texttt{b} \Rightarrow \texttt{b}'\ \texttt{not a function result}}{\texttt{b}(\texttt{a}) \Rightarrow \texttt{wrong}}$$

$$\frac{}{\texttt{nil} \Rightarrow \texttt{nil}}$$

$$\frac{\begin{array}{cc}\texttt{a} \Rightarrow \texttt{a}'\ \texttt{an object result} & \texttt{no f,m in a}' \\ \texttt{b} \Rightarrow \texttt{b}'\ (\neq \texttt{wrong}) & \texttt{c} \Rightarrow \texttt{c}'\ (\neq \texttt{wrong})\end{array}}{\texttt{a}[\texttt{f} = \texttt{b}, \texttt{m} = \texttt{c}] \Rightarrow \texttt{a}'[\texttt{f} = \texttt{b}', \texttt{m} = \texttt{c}']}$$

$$\frac{\texttt{a} \Rightarrow \texttt{a}'\ \texttt{not an object result, or f or m in a}'}{\texttt{a}[\texttt{f} = \texttt{b}, \texttt{m} = \texttt{c}] \Rightarrow \texttt{wrong}}$$

$$\frac{\mathtt{a} \Rightarrow \mathtt{a}' \qquad \mathtt{b} \Rightarrow \mathtt{wrong}}{\mathtt{a}[\mathtt{f} = \mathtt{b}, \mathtt{m} = \mathtt{c}] \Rightarrow \mathtt{wrong}}$$

$$\frac{\mathtt{a} \Rightarrow \mathtt{a}' \qquad \mathtt{b} \Rightarrow \mathtt{b}' \qquad \mathtt{c} \Rightarrow \mathtt{wrong}}{\mathtt{a}[\mathtt{f} = \mathtt{b}, \mathtt{m} = \mathtt{c}] \Rightarrow \mathtt{wrong}}$$

$$\frac{\mathtt{a} \Rightarrow \mathtt{a}' \qquad \mathtt{f} = \mathtt{b}\ \mathtt{in}\ \mathtt{a}'}{\mathtt{a.f} \Rightarrow \mathtt{b}}$$

$$\frac{\mathtt{a} \Rightarrow \mathtt{a}' \qquad \mathtt{m} = \mathtt{c}\ \mathtt{in}\ \mathtt{a}' \qquad \mathtt{c}(\mathtt{a}') \Rightarrow \mathtt{d}}{\mathtt{a.m} \Rightarrow \mathtt{d}}$$

$$\frac{\mathtt{a} \Rightarrow \mathtt{a}' \qquad \mathtt{no}\ \mathtt{f}\ \mathtt{in}\ \mathtt{a}'}{\mathtt{a.f} \Rightarrow \mathtt{wrong}}$$

$$\frac{\mathtt{a} \Rightarrow \mathtt{a}' \qquad \mathtt{no}\ \mathtt{m}\ \mathtt{in}\ \mathtt{a}'}{\mathtt{a.m} \Rightarrow \mathtt{wrong}}$$

$$\frac{\mathtt{a} \Rightarrow \mathtt{a}' \qquad \mathtt{f}\ \mathtt{in}\ \mathtt{a}' \qquad \mathtt{b} \Rightarrow \mathtt{b}'\ (\neq \mathtt{wrong})}{\mathtt{a.f} := \mathtt{b} \Rightarrow \mathtt{a}'\{\mathtt{f} \leftarrow \mathtt{b}'\}}$$

$$\frac{\mathtt{a} \Rightarrow \mathtt{a}' \qquad \mathtt{m}\ \mathtt{in}\ \mathtt{a}' \qquad \mathtt{c} \Rightarrow \mathtt{c}'\ (\mathrm{neq}\mathtt{wrong})}{\mathtt{a.m} := \mathtt{c} \Rightarrow \mathtt{a}'\{\mathtt{m} \leftarrow \mathtt{c}'\}}$$

$$\frac{\mathtt{a} \Rightarrow \mathtt{a}' \qquad \mathtt{no}\ \mathtt{f}\ \mathtt{in}\ \mathtt{a}'}{\mathtt{a.f} := \mathtt{b} \Rightarrow \mathtt{wrong}}$$

$$\frac{\mathtt{a} \Rightarrow \mathtt{a}' \qquad \mathtt{no}\ \mathtt{m}\ \mathtt{in}\ \mathtt{a}'}{\mathtt{a.m} := \mathtt{c} \Rightarrow \mathtt{wrong}}$$

$$\frac{\mathtt{a} \Rightarrow \mathtt{a}' \qquad \mathtt{b} \Rightarrow \mathtt{wrong}}{\mathtt{a.f} := \mathtt{b} \Rightarrow \mathtt{wrong}}$$

$$\frac{\mathtt{a} \Rightarrow \mathtt{a}' \qquad \mathtt{c} \Rightarrow \mathtt{wrong}}{\mathtt{a.m} := \mathtt{c} \Rightarrow \mathtt{wrong}}$$

$$\frac{}{\mathtt{wrong} \Rightarrow \mathtt{wrong}}$$

Discussion

In these rules, we have made some choices of order of evaluation. We believe that all of the choices are reasonable, and they simplify our presentation. In particular, the rules for functions are usual ones for call-by-value reduction. More interestingly, we evaluate fields and methods before they are collected into objects, rather than delay their evaluation until they are accessed. Thus, if $\mathtt{b}$ does not reduce to a result, then neither do $\mathtt{a.f} := \mathtt{b}$ and $\mathtt{a}[\mathtt{f} = \mathtt{b}, \mathtt{m} = \mathtt{c}]$ (unless they reduce to $\mathtt{wrong}$). This seems like the most sensible choice for a call-by-value functional language, particularly with the context of an imperative language in mind. In an imperative language, b may depend on program variables, and these have to be accessed before they change; b can even make reference to a.f.

We have made other choices that cannot be detected in a typed setting. For example, the rules allow storing a non-function c as method of an object a, with a.m := c. An error is produced only if the method is invoked. However, if c is not a function then a.m := c is not typable; thus the possibility allowed by the reduction rules is irrelevant for well-typed programs.

3.4 Subject reduction

With the syntax of Baby Modula-3 complete, we start the study of syntactic properties. We obtain a subject-reduction theorem:

Theorem 1
If $\mathtt{a} \Rightarrow \mathtt{a}'$ and $\vdash \mathtt{a} : \mathtt{A}$ then $\vdash \mathtt{a}' : \mathtt{A}$.

A very typical substitution lemma and some additional syntactic observations are useful in the proof of the theorem:

Lemma 1
If A and B are closed, $\mathtt{E}, \mathtt{x} : \mathtt{B} \vdash \mathtt{a} : \mathtt{A}$, and $\vdash \mathtt{b} : \mathtt{B}$ for a result b, then $\mathtt{E} \vdash \mathtt{a}[\mathtt{b}/\mathtt{x}] : \mathtt{A}$.

Proof
The proof is by induction on the length of a proof of $\mathtt{E}, \mathtt{x} : \mathtt{B} \vdash \mathtt{a} : \mathtt{A}$. The only important cases are those to do with functions, and they are treated abundantly in the literature. □

Proposition 1
If $\mathtt{E} \vdash \mathtt{a} : \mathtt{A}\ \ \mathtt{Self{=}S}$ is derivable, then A is an object type expression of the form $\mathtt{Root}[\mathtt{fi} : \mathtt{Bi}, \mathtt{mj} : \mathtt{Cj}] \ldots [\mathtt{fk} : \mathtt{Bk}, \mathtt{ml} : \mathtt{Cl}]$, and a is a term of the corresponding form $\mathtt{nil}[\mathtt{fi} = \mathtt{bi}, \mathtt{mj} = \mathtt{cj}] \ldots [\mathtt{fk} = \mathtt{bk}, \mathtt{ml} = \mathtt{cl}]$.

Proof
The proof is an easy induction on derivations. □

Now we strengthen the claim of Theorem 1, and prove:

Lemma 2
Assume that $\mathtt{a} \Rightarrow \mathtt{a}'$.

- If $\vdash \mathtt{a} : \mathtt{A}$ then $\vdash \mathtt{a}' : \mathtt{A}$.
- If $\vdash \mathtt{a} : \mathtt{A}\ \ \mathtt{Self{=}S}$ then $\vdash \mathtt{a}' : \mathtt{A}\ \ \mathtt{Self{=}S}$.

Proof
The proof is by induction on the length of the reduction derivation. The cases for reductions to **wrong** are vacuously true; we treat only one of them, as an example. Also, by Proposition 1, $\vdash \mathtt{a} : \mathtt{A}\ \ \mathtt{Self{=}S}$ can hold only if a is built from **nil** by extension, and so we consider the second part of the claim only in the appropriate cases. Finally, we include only cases for the object constructs, the other ones being standard.

In most of this proof, we work with types up to provable equality.

- The case of $\texttt{nil} \Rightarrow \texttt{nil}$ is trivial.
- Suppose that $\vdash \texttt{a}[\texttt{f} = \texttt{b}, \texttt{m} = \texttt{c}] : \texttt{D}$ and consider the rule:

$$\frac{\texttt{a} \Rightarrow \texttt{a}' \ \texttt{an object result} \qquad \texttt{no f,m in a}' \\ \texttt{b} \Rightarrow \texttt{b}' \ (\neq \texttt{wrong}) \qquad \texttt{c} \Rightarrow \texttt{c}' \ (\neq \texttt{wrong})}{\texttt{a}[\texttt{f} = \texttt{b}, \texttt{m} = \texttt{c}] \Rightarrow \texttt{a}'[\texttt{f} = \texttt{b}', \texttt{m} = \texttt{c}']}$$

  The assumption implies that for some $\texttt{D}'$ we have $\vdash \texttt{a}[\texttt{f} = \texttt{b}, \texttt{m} = \texttt{c}] : \texttt{D}'$ and $\vdash \texttt{a}[\texttt{f} = \texttt{b}, \texttt{m} = \texttt{c}] : \texttt{D}' \ \ \texttt{Self=D}'$, with $\texttt{D}'$ (provably equal to) a subtype of $\texttt{D}$ (possibly $\texttt{D}$ itself). By Proposition 1 it follows that $\texttt{D}'$ has the form $\texttt{A}[\texttt{f} : \texttt{B}, \texttt{m} : \texttt{C}]$ and $\vdash \texttt{a} : \texttt{A} \ \ \texttt{Self=D}'$. Moreover, it follows that $\vdash \texttt{b} : \texttt{B}$ and $\vdash \texttt{c} : \texttt{G} {\to} \texttt{C}$ for some $\texttt{G}$ with $\vdash \texttt{D}' \leq \texttt{G}$. By induction hypothesis, $\vdash \texttt{a}' : \texttt{A} \ \ \texttt{Self=D}'$, $\vdash \texttt{b}' : \texttt{B}$, and $\vdash \texttt{c}' : \texttt{G} {\to} \texttt{C}$. Hence, it follows that $\vdash \texttt{a}'[\texttt{f} = \texttt{b}', \texttt{m} = \texttt{c}'] : \texttt{D}' \ \ \texttt{Self=D}'$, and so that $\vdash \texttt{a}'[\texttt{f} = \texttt{b}', \texttt{m} = \texttt{c}'] : \texttt{D}'$. Since $\vdash \texttt{D}' \leq \texttt{D}$, subsumption yields the result $\vdash \texttt{a}'[\texttt{f} = \texttt{b}', \texttt{m} = \texttt{c}'] : \texttt{D}$.
  In this case, because of the form of the terms involved, the second half of the subject-reduction claim is relevant. For this second half, we need to prove that $\vdash \texttt{a}[\texttt{f} = \texttt{b}, \texttt{m} = \texttt{c}] : \texttt{D} \ \ \texttt{Self=S}$ implies $\vdash \texttt{a}'[\texttt{f} = \texttt{b}', \texttt{m} = \texttt{c}'] : \texttt{D} \ \ \texttt{Self=S}$. While this proof is not trivial, it is a simple variant of the one just given.
- Suppose that $\vdash \texttt{a.f} : \texttt{B}$ and consider the rule:

$$\frac{\texttt{a} \Rightarrow \texttt{a}' \qquad \texttt{f} = \texttt{b} \ \texttt{in a}'}{\texttt{a.f} \Rightarrow \texttt{b}}$$

  Inspection of the type rules shows that $\vdash \texttt{a} : \texttt{A}$ for some object type $\texttt{A}$ and, further, $\texttt{f} : \texttt{B}' \ \texttt{in A}$ with $\texttt{B}'$ (provably equal to) some subtype of $\texttt{B}$ (possibly $\texttt{B}$ itself). By induction hypothesis, $\vdash \texttt{a}' : \texttt{A}$; since $\texttt{a}'$ is an object result and $\texttt{f} : \texttt{B}' \ \texttt{in A}$, the proof of $\vdash \texttt{a}' : \texttt{A}$ must involve a proof of $\vdash \texttt{b} : \texttt{B}'$, and by subsumption we can obtain $\vdash \texttt{b} : \texttt{B}$.
- Suppose that $\vdash \texttt{a.m} : \texttt{C}$ and consider the rule:

$$\frac{\texttt{a} \Rightarrow \texttt{a}' \qquad \texttt{m} = \texttt{c} \ \texttt{in a}' \qquad \texttt{c}(\texttt{a}') \Rightarrow \texttt{d}}{\texttt{a.m} \Rightarrow \texttt{d}}$$

  As in the previous case, $\vdash \texttt{a} : \texttt{A}$ for some object type $\texttt{A}$ and, further, $\texttt{m} : \texttt{C}' \ \texttt{in A}$ where $\texttt{C}'$ is provably equal to a subtype of $\texttt{C}$ (possibly $\texttt{C}$ itself). By induction hypothesis, $\vdash \texttt{a}' : \texttt{A}$. Now it follows that for some $\texttt{A}'$ and $\texttt{D}$, $\vdash \texttt{c} : \texttt{D} {\to} \texttt{C}'$, $\vdash \texttt{A}' \leq \texttt{D}$, $\vdash \texttt{A}' \leq \texttt{A}$, and $\vdash \texttt{a}' : \texttt{A}'$. Subsumption yields $\vdash \texttt{a}' : \texttt{D}$, and the type-checking rule for application yields $\vdash \texttt{c}(\texttt{a}') : \texttt{C}'$. By induction hypothesis, $\vdash \texttt{d} : \texttt{C}'$. Subsumption finally yields $\vdash \texttt{d} : \texttt{C}$, as desired.
- Suppose that $\vdash \texttt{a.f} : \texttt{B}$ and consider the rule:

$$\frac{\texttt{a} \Rightarrow \texttt{a}' \qquad \texttt{no f in a}'}{\texttt{a.f} \Rightarrow \texttt{wrong}}$$

  As in the case where the value of a field is read without error, we obtain that $\vdash \texttt{a}' : \texttt{A}$ for some object type $\texttt{A}$ and $\texttt{f} : \texttt{B}' \ \texttt{in A}$ for some $\texttt{B}'$ such that $\vdash \texttt{B}' \leq \texttt{B}$. It is easy to see that this contradicts the assumption $\texttt{no f in a}'$.
- Suppose that $\vdash \texttt{a.f} := \texttt{b} : \texttt{A}$ and consider the rule:

$$\frac{\texttt{a} \Rightarrow \texttt{a}' \qquad \texttt{f in a}' \qquad \texttt{b} \Rightarrow \texttt{b}' \ (\neq \texttt{wrong})}{\texttt{a.f} := \texttt{b} \Rightarrow \texttt{a}'\{\texttt{f} \leftarrow \texttt{b}'\}}$$

Much as in other cases, it must be that, for some $\mathtt{A}'$ and $\mathtt{B}$, $\vdash\ \mathtt{a} : \mathtt{A}'$, $\vdash\ \mathtt{A}' \leq \mathtt{A}$, $\mathtt{f} : \mathtt{B}\ \mathtt{in}\ \mathtt{A}'$, and $\vdash\ \mathtt{b} : \mathtt{B}$. By induction hypothesis, $\vdash\ \mathtt{a}' : \mathtt{A}'$ and $\vdash\ \mathtt{b}' : \mathtt{B}$. Now a proof that $\vdash\ \mathtt{a}'\{\mathtt{f} \leftarrow \mathtt{b}'\} : \mathtt{A}'$ can be obtained from the proof that $\vdash\ \mathtt{a}' : \mathtt{A}'$ by replacing the typing proof for the $\mathtt{f}$ field of $\mathtt{a}'$ with the proof of $\vdash\ \mathtt{b}' : \mathtt{B}$. The proof that $\vdash\ \mathtt{a}'\{\mathtt{f} \leftarrow \mathtt{b}'\} : \mathtt{A}$ follows by subsumption.

- Suppose that $\vdash\ \mathtt{a.m} := \mathtt{c} : \mathtt{A}$ and consider the rule:

$$\frac{\mathtt{a}\ \Rightarrow\ \mathtt{a}' \qquad \mathtt{m}\ \mathtt{in}\ \mathtt{a}' \qquad \mathtt{c}\ \Rightarrow\ \mathtt{c}'\ \ (\neq \mathtt{wrong})}{\mathtt{a.m} := \mathtt{c}\ \Rightarrow\ \mathtt{a}'\{\mathtt{m} \leftarrow \mathtt{c}'\}}$$

In the present case, it must be that, for some $\mathtt{A}'$, $\mathtt{C}$, and $\mathtt{D}$, $\vdash\ \mathtt{a} : \mathtt{A}'$, $\vdash\ \mathtt{A}' \leq \mathtt{A}$, $\mathtt{m} : \mathtt{C}\ \mathtt{in}\ \mathtt{A}'$, $\vdash\ \mathtt{c} : \mathtt{D}{\rightarrow}\mathtt{C}$, and $\vdash\ \mathtt{A}' \leq \mathtt{D}$. By induction hypothesis, $\vdash\ \mathtt{a}' : \mathtt{A}'$ and $\vdash\ \mathtt{c}' : \mathtt{D}{\rightarrow}\mathtt{C}$. A proof that $\vdash\ \mathtt{a}'\{\mathtt{m} \leftarrow \mathtt{c}'\} : \mathtt{A}'$ can be obtained from the proof that $\vdash\ \mathtt{a}' : \mathtt{A}'$ by replacing the typing proof for the $\mathtt{m}$ method of $\mathtt{a}'$ with the proof of $\vdash\ \mathtt{c}' : \mathtt{D}{\rightarrow}\mathtt{C}$. The proof that $\vdash\ \mathtt{a}'\{\mathtt{f} \leftarrow \mathtt{b}'\} : \mathtt{A}$ follows by subsumption.

$\square$

Since the type rules do not give any type for $\mathtt{wrong}$, it follows:

Corollary 1
If $\mathtt{a}\ \Rightarrow\ \mathtt{a}'$ and $\vdash\ \mathtt{a} : \mathtt{A}$ then $\mathtt{a}'$ is not $\mathtt{wrong}$.

4 Semantics

This section concerns the denotational semantics of Baby Modula-3. Subsection 4.1 is about the untyped semantics of the terms of the language; this part is relatively straightforward, although it involves a few subtle choices. Subsection 4.2, which is harder, gives a semantics for the type system. Subsection 4.3 is a short discussion of program verification. Subsection 4.4 briefly describes a second, stronger semantics for the type system.

4.1 Semantics of terms

We interpret the language in an untyped model. After describing this untyped model in the first subsection, we define the interpretation of the terms of Baby Modula-3. Subsection 4.1.3 relates this interpretation with the reduction rules of Section 3. The preliminary material on the untyped model is rather technical. It contains details not necessary for understanding most of the rest of the paper.

4.1.1 Preliminaries

The underlying assumptions on the untyped model are much as in (MacQueen *et al.*, 1986); we assume a complete partial order $(D, \sqsubseteq)$ such that:

- There is an increasing sequence $p_n : D{\rightarrow}D$ of continuous projections with least upper bound the identity. Further, p_0 constantly equals $\perp$.

- There are strict, continuous embedding-retraction pairs (e, r) between D and each of O, $(D \to D)_\perp$, and $(L \to D)$.

$$\begin{array}{rcl} O \xrightarrow{e} & D & \xrightarrow{r} O \\ (D \to D)_\perp \xrightarrow{e} & D & \xrightarrow{r} (D \to D)_\perp \\ (L \to D) \xrightarrow{e} & D & \xrightarrow{r} (L \to D) \end{array}$$

  Here O is a two-point partial order $\{*\}_\perp$; we view $*$ as the error value. As usual $(D \to D)$ is the complete partial order of continuous functions from D to D, and $(D \to D)_\perp$ is its lifting. (The importance of lifting is discussed further below.) Finally, L is a set of labels $\{\mathtt{f0}, \mathtt{f1}, \ldots, \mathtt{m0}, \mathtt{m1}, \ldots\}$, and the summand $(L \to D)$ can be viewed, roughly, as the set of records over these labels. This summand is essentially a product (of D over L), and we can make this product strict or not. Having a strict product amounts to identifying all elements that map any label to $\perp$, and interpreting them all as $\perp$; a non-strict product keeps these elements separate. A strict semantics corresponds better to our reduction rules and is closer to full abstraction, while a non-strict semantics affords us more flexibility. The definitions below can be read with either choice.
  We omit the various e's and r's in most of what follows, and do not distinguish them. We view O, $(D \to D)$, and $(L \to D)$ as subsets of D.
- Let ; denote function composition, so that $(x; y)(z) = y(x(z))$. For all i,

$$\begin{array}{ll} p_{i+1}(e(*)) = e(*) & \\ p_{i+1}(e(f)) = e(p_i; f; p_i) & f \in D \to D \\ p_{i+1}(e(o)) = e(o; p_i) & o \in L \to D \end{array}$$

An element v of D is finite if $v \sqsubseteq \sqcup_k \langle u_k \rangle$ implies $v \sqsubseteq u_i$ for some i, and the least n for which $p_n(v) = v$ is the rank of v.

To obtain D, one can solve an appropriate domain equation, such as:

$$D \;=\; O + (D \to D)_\perp + (L \to D)$$

by the usual "limit of a sequence of iterates" method.

4.1.2 Definitions

We define the semantics function for terms:

$$[\![\ ]\!] : \ (V \to D) \to (E \to D)$$

where V is the set of variables and E the set of expressions. We call a mapping ρ in $V \to D$ an environment and write $[\![\mathtt{a}]\!]_\rho$ for the semantics of a term $\mathtt{a}$ with an environment ρ. When $\mathtt{a}$ is closed, we may write $[\![\mathtt{a}]\!]$, omitting ρ since it is irrelevant.

If f is a function (for example, an environment) and $\mathtt{l}$ is in its domain, we write $f\{\mathtt{l} \leftarrow v\}$ for the function that maps $\mathtt{l}$ to v and is otherwise identical to f. (The same notation is used for a different but related notion in Subsection 3.3.)

We set:

$$[\![\mathtt{x}]\!]_\rho \;=\; \rho(\mathtt{x})$$

$$
\begin{array}{rcl}
[\![\mathtt{fun(x:A)b}]\!]_\rho & = & \lambda v.([\![\mathtt{b}]\!]_{\rho\{\mathtt{x}\leftarrow v\}}) \\
[\![\mathtt{b(a)}]\!]_\rho & = & \text{if } [\![\mathtt{a}]\!]_\rho \neq * \text{ and } [\![\mathtt{b}]\!]_\rho \in (D{\rightarrow}D) \\
 & & \text{then } [\![\mathtt{b}]\!]_\rho([\![\mathtt{a}]\!]_\rho) \\
 & & \text{else } * \\
[\![\mathtt{nil}]\!]_\rho & = & \text{the constantly } * \text{ function in } L{\rightarrow}D \\
[\![\mathtt{a[f=b,m=c]}]\!]_\rho & = & \text{if } [\![\mathtt{a}]\!]_\rho \in (L{\rightarrow}D),\ [\![\mathtt{a}]\!]_\rho(\mathtt{f}) = *,\ [\![\mathtt{a}]\!]_\rho(\mathtt{m}) = *, \\
 & & [\![\mathtt{b}]\!]_\rho \neq *, \text{ and } [\![\mathtt{c}]\!]_\rho \neq * \\
 & & \text{then } [\![\mathtt{a}]\!]_\rho\{\mathtt{f} \leftarrow [\![\mathtt{b}]\!]_\rho\}\{\mathtt{m} \leftarrow [\![\mathtt{c}]\!]_\rho\} \\
 & & \text{else } * \\
[\![\mathtt{a.f}]\!]_\rho & = & \text{if } [\![\mathtt{a}]\!]_\rho \in (L{\rightarrow}D) \text{ then } [\![\mathtt{a}]\!]_\rho(\mathtt{f}) \text{ else } * \\
[\![\mathtt{a.m}]\!]_\rho & = & \text{if } [\![\mathtt{a}]\!]_\rho \in (L{\rightarrow}D) \text{ and } [\![\mathtt{a}]\!]_\rho(\mathtt{m}) \in (D{\rightarrow}D) \\
 & & \text{then } [\![\mathtt{a}]\!]_\rho(\mathtt{m})([\![\mathtt{a}]\!]_\rho) \\
 & & \text{else } * \\
[\![\mathtt{a.f:=b}]\!]_\rho & = & \text{if } [\![\mathtt{a}]\!]_\rho \in (L{\rightarrow}D),\ [\![\mathtt{a}]\!]_\rho(\mathtt{f}) \neq *, \text{ and } [\![\mathtt{b}]\!]_\rho \neq * \\
 & & \text{then } [\![\mathtt{a}]\!]_\rho\{\mathtt{f} \leftarrow [\![\mathtt{b}]\!]_\rho\} \\
 & & \text{else } * \\
[\![\mathtt{a.m:=c}]\!]_\rho & = & \text{if } [\![\mathtt{a}]\!]_\rho \in (L{\rightarrow}D),\ [\![\mathtt{a}]\!]_\rho(\mathtt{m}) \neq *, \text{ and } [\![\mathtt{c}]\!]_\rho \neq * \\
 & & \text{then } [\![\mathtt{a}]\!]_\rho\{\mathtt{m} \leftarrow [\![\mathtt{c}]\!]_\rho\} \\
 & & \text{else } * \\
[\![\mathtt{wrong}]\!]_\rho & = & *
\end{array}
$$

This definition is given in a metalanguage where conjunctions and conditionals are strict and evaluated left to right, and $\in$ is a strict membership test. For example, if $[\![\mathtt{a}]\!]_\rho = \bot$ then immediately $[\![\mathtt{a[f=b,m=c]}]\!]_\rho = \bot$.

4.1.3 Soundness

The main theorem about the term interpretation states the soundness of reduction. This theorem says that reduction does not change the meaning of programs:

Theorem 2
If $\mathtt{a}$ and $\mathtt{b}$ are closed and $\mathtt{a} \Rightarrow \mathtt{b}$ then $[\![\mathtt{a}]\!] = [\![\mathtt{b}]\!]$.

Proof
The proof is by induction on the derivation of $\mathtt{a} \Rightarrow \mathtt{b}$. It relies on the observations:

- If $\mathtt{d}$ is a result then $[\![\mathtt{d}]\!] \neq \bot$.
- If $\mathtt{d}$ is a result then it is a proper result if and only if $[\![\mathtt{d}]\!] \neq *$.
- If $\mathtt{d}$ is a result then it is a function result if and only if $[\![\mathtt{d}]\!] \in (D{\rightarrow}D)$.
- If $\mathtt{d}$ is a proper result then $[\![\mathtt{(fun(x:A)b)(d)}]\!] = [\![\mathtt{fun(x:A)b}]\!]([\![\mathtt{d}]\!]) = [\![\mathtt{b[d/x]}]\!]$.
- If $\mathtt{d}$ is a result then it is an object result if and only if $[\![\mathtt{d}]\!] \in (L{\rightarrow}D)$.
- If $\mathtt{d}$ is an object result and $\mathtt{f = b\ in\ d}$ then $[\![\mathtt{d}]\!](\mathtt{f}) = [\![\mathtt{b}]\!]$.
- If $\mathtt{d}$ is an object result and $\mathtt{m = c\ in\ d}$ then $[\![\mathtt{d}]\!](\mathtt{m}) = [\![\mathtt{c}]\!]$.

- If $\mathtt{d}$ is an object result with $\mathtt{no\ f\ in\ d}$ then $[\![\mathtt{d}]\!](\mathtt{f}) = *$.
- If $\mathtt{d}$ is an object result with $\mathtt{no\ m\ in\ d}$ then $[\![\mathtt{d}]\!](\mathtt{m}) = *$.
- If $\mathtt{d}$ is an object result with $\mathtt{no\ f, m\ in\ d}$, and $\mathtt{b}$ and $\mathtt{c}$ are proper results, then $[\![\mathtt{d}[\mathtt{f} = \mathtt{b}, \mathtt{m} = \mathtt{c}]]\!] = [\![\mathtt{d}]\!]\{\mathtt{f} \leftarrow [\![\mathtt{b}]\!]\}\{\mathtt{m} \leftarrow [\![\mathtt{c}]\!]\}$.
- If $\mathtt{d}$ is an object result with $\mathtt{f\ in\ d}$ and $\mathtt{b}$ is a proper result, then $[\![\mathtt{d}\{\mathtt{f} \leftarrow \mathtt{b}\}]\!] = [\![\mathtt{d}]\!]\{\mathtt{f} \leftarrow [\![\mathtt{b}]\!]\}$.
- If $\mathtt{d}$ is an object result with $\mathtt{m\ in\ d}$ and $\mathtt{c}$ is a proper result, then $[\![\mathtt{d}\{\mathtt{m} \leftarrow \mathtt{c}\}]\!] = [\![\mathtt{d}]\!]\{\mathtt{m} \leftarrow [\![\mathtt{c}]\!]\}$.

With these observations, we treat two cases as examples:

- Suppose that $\mathtt{a.f}$ reduces to $\mathtt{b}$ using the rule:
$$\frac{\mathtt{a} \Rightarrow \mathtt{a}' \qquad \mathtt{f} = \mathtt{b\ in\ a}'}{\mathtt{a.f} \Rightarrow \mathtt{b}}$$
Since $\mathtt{a}'$ is a result, $[\![\mathtt{a}']\!] \neq \bot$. Since $\mathtt{f} = \mathtt{b\ in\ a}'$, $\mathtt{a}'$ is an object result, so $[\![\mathtt{a}']\!] \in (L \to D)$, and $[\![\mathtt{a}']\!](\mathtt{f}) = [\![\mathtt{b}]\!]$. The induction hypothesis $[\![\mathtt{a}]\!] = [\![\mathtt{a}']\!]$ together with the definition of $[\![\ ]\!]$ yields that $[\![\mathtt{a.f}]\!] = [\![\mathtt{b}]\!]$.
- Suppose that $\mathtt{a.f}$ reduces to $\mathtt{wrong}$ using the rule:
$$\frac{\mathtt{a} \Rightarrow \mathtt{a}' \qquad \mathtt{no\ f\ in\ a}'}{\mathtt{a.f} \Rightarrow \mathtt{wrong}}$$
Since $\mathtt{a}'$ is a result, $[\![\mathtt{a}']\!] \neq \bot$. Since $\mathtt{no\ f\ in\ a}'$, $[\![\mathtt{a}']\!](\mathtt{f}) = *$. The induction hypothesis $[\![\mathtt{a}]\!] = [\![\mathtt{a}']\!]$ together with the definition of $[\![\ ]\!]$ yields that $[\![\mathtt{a.f}]\!] = *$, that is, $[\![\mathtt{a.f}]\!] = [\![\mathtt{wrong}]\!]$.
- Suppose that $\mathtt{a.m}$ reduces to $\mathtt{d}$ using the rule:
$$\frac{\mathtt{a} \Rightarrow \mathtt{a}' \qquad \mathtt{m} = \mathtt{c\ in\ a}' \qquad \mathtt{c}(\mathtt{a}') \Rightarrow \mathtt{d}}{\mathtt{a.m} \Rightarrow \mathtt{d}}$$
Since $\mathtt{a}'$ is a result, $[\![\mathtt{a}']\!] \neq \bot$. Since $\mathtt{m} = \mathtt{c\ in\ a}'$, $\mathtt{a}'$ is an object result, so $[\![\mathtt{a}']\!] \in (L \to D)$, and $[\![\mathtt{a}']\!](\mathtt{m}) = [\![\mathtt{c}]\!]$. Moreover, since $\mathtt{c}$ is a result, $[\![\mathtt{c}]\!] \neq \bot$. The rest of the proof is by cases:
  - — Assume that $[\![\mathtt{c}]\!] \in (D \to D)$. The induction hypothesis $[\![\mathtt{a}]\!] = [\![\mathtt{a}']\!]$ together with the definition of $[\![\ ]\!]$ yields that $[\![\mathtt{a.m}]\!] = [\![\mathtt{a}]\!](\mathtt{m})([\![\mathtt{a}]\!])$, that is, $[\![\mathtt{a.m}]\!] = [\![\mathtt{c}]\!]([\![\mathtt{a}']\!])$. Since $[\![\mathtt{a}']\!] \in (L \to D)$, $[\![\mathtt{c}]\!]([\![\mathtt{a}']\!]) = [\![\mathtt{c}(\mathtt{a}')]\!]$, and the induction hypothesis $[\![\mathtt{c}(\mathtt{a}')]\!] = [\![\mathtt{d}]\!]$ yields that $[\![\mathtt{a.m}]\!] = [\![\mathtt{d}]\!]$.
  - — Assume that $[\![\mathtt{c}]\!] \notin (D \to D)$. The induction hypothesis $[\![\mathtt{a}]\!] = [\![\mathtt{a}']\!]$ together with the definition of $[\![\ ]\!]$ yields that $[\![\mathtt{a.m}]\!] = *$. Moreover, $[\![\mathtt{c}(\mathtt{a}')]\!] = *$ because $[\![\mathtt{a}']\!] \neq \bot$, and the induction hypothesis $[\![\mathtt{c}(\mathtt{a}')]\!] = [\![\mathtt{d}]\!]$ yields that $[\![\mathtt{d}]\!] = *$ as well.

□

It is now easy to see why we take $(D \to D)_\bot$ rather than $(D \to D)$ in the definition of the semantic domain. The lifting leaves room for a least function different from D's $\bot$; this means that we can define the meaning of a result $\mathtt{fun}(\mathtt{x} : \mathtt{A})\mathtt{b}$ to differ from $\bot$ even if $\mathtt{b}$ is constantly $\bot$. We take advantage of this freedom. This choice is embodied in the reduction rules, where evaluation does not go under function binders; it is reflected in the denotational semantics; and it is then essential for the proof that the reduction rules are correct with respect to the denotational semantics.

On the other hand, we do not take $(L \to D)_\perp$ instead of $(L \to D)$. The extra flexibility obtained by lifting $(L \to D)$, while not problematic, is unnecessary because we adopt a strict semantics of objects (so for example $\texttt{nil}[\texttt{f} = \texttt{b}, \texttt{m} = \texttt{c}]$ denotes $\perp$ if `b` does). The treatment of a non-strict language may be a good exercise.

It would be worthwhile to study the semantics further, and in particular to consider issues beyond soundness, such as adequacy and full abstraction. We postpone the study of these issues.

4.2 Semantics of types

Having given the semantics in an untyped model, we view the types as certain subsets of this untyped model. These subsets are ideals (MacQueen *et al.*, 1986).

Ideals suffice for our purpose—studying type rules. However, they do not yield a proper model of typed lambda calculi, because they do not validate one of the standard equational rules for typed lambda calculi, the ξ rule (see for example (Gunter, 1992, pp. 44, 265)). We discuss the ξ rule and alternatives to ideals in Subsection 4.4.

After some preliminaries, we define the ideal interpretation and then use it to prove the soundness of the type rules of Baby Modula-3. As in the previous subsection, the preliminaries are rather technical; the details are not essential for an intuitive understanding of the main definitions below.

4.2.1 Preliminaries

An ideal is a subset I of D with the properties:

- I is nonempty;
- I is closed downwards in the $\sqsubseteq$ order;
- I is closed under limits of increasing sequences in the $\sqsubseteq$ order.

We write **Idl** for the set of all ideals that do not contain $*$. By convention, the variables R, T, Rd, Te, $\ldots$, and S range over **Idl**. In the next subsection, all types are interpreted as ideals in **Idl**.

The distance between two ideals is 2^{-r}, where r is the minimum rank of the elements in one ideal but not the other, and it is 0 if the two ideals are equal. The set of all ideals with this distance function is a complete metric space, and so is **Idl**. Furthermore, by the Banach Fixpoint Theorem, if F is a contractive (distance-reducing) map between ideals then it has a unique fixpoint; and if it maps **Idl** to **Idl**, then the fixpoint is in **Idl** as well. This is the basis of the usual interpretation of recursive types.

4.2.2 Definitions

In this section, we define the semantics function for types:

$$[\![\ ]\!] : (TV \to \mathbf{Idl}) \to (TE \to \mathbf{Idl})$$

where TV is the set of type variables and TE the set of type expressions. A mapping ρ in $TV \to \mathbf{Idl}$ is a type environment, and we write $[\![\mathtt{A}]\!]_\rho$ for the semantics of a type A with the environment ρ. By definition, $[\![\mathtt{X}]\!]_\rho = \rho(\mathtt{X})$. For convenience, we merge environments and type environments, and call environments the functions in $(V \to D) \cap (TV \to \mathbf{Idl})$.

The relation $\leq$ is simply interpreted as ideal containment. The function-space operator is given by:

$$R \to T = \{\bot\} \cup \{f \in (D \to D) \mid f(R) \subseteq T\}$$

and we set:

$$[\![\mathtt{A} \to \mathtt{B}]\!]_\rho = [\![\mathtt{A}]\!]_\rho \to [\![\mathtt{B}]\!]_\rho$$

It turns out to be useful to interpret expressions of the form `A Self=S` as ordinary types. Intuitively, `A Self=S` is much like the object type A, but the self-application present in the semantics of A objects is replaced with an application to an element of S. Thus, `A Self=S` is essentially a record type. For example, if $\mathtt{A} = [\mathtt{f} : \mathtt{Nat}, \mathtt{m} : \mathtt{Nat}]$ then `A Self=S` can be seen as the type $\langle\!\langle \mathtt{f} : \mathtt{Nat}, \mathtt{m} : \mathtt{S} \to \mathtt{Nat} \rangle\!\rangle$, the type of all records with a field f of type Nat and a field m of type S→Nat.

The definition of $[\![\mathtt{A\ \ Self{=}S}]\!]_\rho$ assumes that A is an object type expression. It relies on two auxiliary functions:

- $\langle \mathtt{A} \rangle^S_\rho$ is an obvious generalization of $[\![\mathtt{A\ \ Self{=}S}]\!]_\rho$, since we set:

  $$[\![\mathtt{A\ \ Self{=}S}]\!]_\rho = \langle \mathtt{A} \rangle^{[\![\mathtt{S}]\!]_\rho}_\rho$$

  with

  $$\langle\ \rangle : \mathbf{Idl} \to (TV \to \mathbf{Idl}) \to (TE \to \mathbf{Idl})$$

- Given a list of ideals R, T, ... and a list of labels f, m, ... of equal length, $\mathcal{R}^S(\mathtt{f} : R, \mathtt{m} : T, \ldots)$ is the set of objects that map f to values in R, m to functions from S to T, ... This is a semantic version of the record type that, informally, can be written $\langle\!\langle \mathtt{f} : R, \mathtt{m} : S \to T, \ldots \rangle\!\rangle$.

The auxiliary functions are defined by:

$$\begin{array}{rcl}
\langle \mathtt{Root} \rangle^S_\rho & = & (L \to D) \\
\langle \mathtt{A}[\mathtt{f} : \mathtt{B}, \mathtt{m} : \mathtt{C}] \rangle^S_\rho & = & \langle \mathtt{A} \rangle^S_\rho \cap \mathcal{R}^S(\mathtt{f} : [\![\mathtt{B}]\!]_\rho, \mathtt{m} : [\![\mathtt{C}]\!]_\rho) \\
\langle \mathtt{Mu(X)A} \rangle^S_\rho & = & \langle \mathtt{A} \rangle^S_{\rho\{\mathtt{X} \leftarrow [\![\mathtt{Mu(X)A}]\!]_\rho\}} \\
\mathcal{R}^S(\mathtt{f} : R, \mathtt{m} : T, \ldots) & = & \{o \in (L \to D) \mid o(\mathtt{f}) \in R \wedge o(\mathtt{m}) \in (S \to T) \wedge \ldots\}
\end{array}$$

The definition is by induction on the size of type expressions. The reference to $[\![\ ]\!]$, whose definition is not yet complete, is justified below.

As suggested in the overview, $\mu(S)\langle \mathtt{A} \rangle^S_\rho$ provides a reasonable interpretation for the object type A. This interpretation does not validate the rules for subtyping, but a simple variant does. Denoting the lub operation on ideals by $\bigsqcup$, we define:

$$\begin{array}{rcl}
[\![\mathtt{A}]\!]_\rho & = & \bigsqcup\{\mu(S)(\langle \mathtt{A} \rangle^S_\rho \cap \mathcal{R}^S(\mathtt{fd} : Rd, \mathtt{me} : Te, \ldots)) \mid \\
& & \qquad \mathtt{fd}, \mathtt{me}, \ldots \text{ a finite list of distinct labels not in } \mathtt{A}\}
\end{array}$$

Roughly, $[\![\mathtt{A}]\!]_\rho$ can be understood as the union of $\mu(S)\langle\mathtt{B}\rangle^S_\rho$ over all extensions $\mathtt{B}$ of $\mathtt{A}$. It should be intuitively clear that this new interpretation is forced to validate the rules for subtyping, and we prove that it validates all other rules as well.

The form of the semantic definition

The functions $\langle\ \rangle$ and $[\![\ ]\!]$ are defined jointly by induction on the size of type expressions. In each case, $\langle\mathtt{A}\rangle$ and $[\![\mathtt{A}]\!]$ are defined in terms of $\langle\mathtt{A}'\rangle$ and $[\![\mathtt{A}']\!]$ for $\mathtt{A}'$ smaller than $\mathtt{A}$, with the exception of the case where $\mathtt{A}$ is a recursive type. In that case, the expression for $\langle\mathtt{Mu(X)A}\rangle$ refers to $[\![\mathtt{Mu(X)A}]\!]$ and vice versa. We have:

$$\begin{aligned}
[\![\mathtt{Mu(X)A}]\!]_\rho &= \bigcup\{\mu(S)(\langle\mathtt{Mu(X)A}\rangle^S_\rho \cap \mathcal{R}^S(\mathtt{fd}: Rd, \mathtt{me}: Te, \ldots)) \mid \\
&\qquad \mathtt{fd}, \mathtt{me}, \ldots \text{ a finite list of distinct labels not in } \mathtt{A}\} \\
&= \bigcup\{\mu(S)(\langle\mathtt{A}\rangle^S_{\rho\{\mathtt{X}\leftarrow[\![\mathtt{Mu(X)A}]\!]_\rho\}} \cap \mathcal{R}^S(\mathtt{fd}: Rd, \mathtt{me}: Te, \ldots)) \mid \\
&\qquad \mathtt{fd}, \mathtt{me}, \ldots \text{ a finite list of distinct labels not in } \mathtt{A}\}
\end{aligned}$$

but this is equivalent to:

$$\begin{aligned}
[\![\mathtt{Mu(X)A}]\!]_\rho &= \mu(T)\bigcup\{\mu(S)(\langle\mathtt{A}\rangle^S_{\rho\{\mathtt{X}\leftarrow T\}} \cap \mathcal{R}^S(\mathtt{fd}: Rd, \mathtt{me}: Te, \ldots)) \mid \\
&\qquad \mathtt{fd}, \mathtt{me}, \ldots \text{ a finite list of distinct labels not in } \mathtt{A}\} \\
&= \mu(T)[\![\mathtt{A}]\!]_{\rho\{\mathtt{X}\leftarrow T\}}
\end{aligned}$$

This reformulation removes the apparent circularity in the definition of $\langle\ \rangle$ and $[\![\ ]\!]$.

The definition of $[\![\ ]\!]$ for object types relies on the existence of certain fixpoints. Corollaries 2 and 3 guarantee the existence of these fixpoints, and also say that they are included in $L\to D$.

Finally, note that $[\![\mathtt{A}]\!]_\rho$ cannot simply be defined as the union of $\mu(S)\langle\mathtt{B}\rangle^S_\rho$ over all extensions $\mathtt{B}$ of $\mathtt{A}$. That appealing definition would be circular, because an extension $\mathtt{B}$ may mention $\mathtt{A}$, like $\mathtt{A}[\mathtt{f}: \mathtt{A}, \mathtt{m}: \mathtt{A}]$, and then $\mu(S)\langle\mathtt{B}\rangle^S_\rho$ would itself be defined in terms of $[\![\mathtt{A}]\!]_\rho$.

4.2.3 Soundness

Next we check the soundness of the type rules with respect to our interpretation. We start with a number of propositions that simplify the argument.

Basic properties of $\langle\ \rangle$

Proposition 2
For all S, $\mathtt{f}$, R, $\mathtt{m}$, T, ..., $\mathcal{R}^S(\mathtt{f}: R, \mathtt{m}: T, \ldots) \subseteq (L\to D)$.

Proof
This follows directly from the definition of $\mathcal{R}^S(\mathtt{f}: R, \mathtt{m}: T, \ldots)$. □

Proposition 3
For all object type expressions $\mathtt{A}$, all S, and all ρ, $\langle\mathtt{A}\rangle^S_\rho \subseteq (L\to D)$.

Proof
The argument is an easy induction on the structure of `A` (more precisely, on the structure of a proof that `A` is an object type expression). □

Proposition 4
For all $\mathtt{f}$, R, $\mathtt{m}$, T, ..., $\mathcal{R}^S(\mathtt{f}:R,\mathtt{m}:T,\ldots)$ is contractive in S and in R, T, ...

Proof
First we check the claim for S. For $\mathcal{R}^S(\mathtt{f}:R)$, this holds because $\mathcal{R}^S(\mathtt{f}:R)$ does not depend on S. For $\mathcal{R}^S(\mathtt{m}:T)$, this follows from the contractiveness of $\rightarrow$ (proved in (MacQueen *et al.*, 1986)), since $\mathcal{R}^S(\mathtt{m}:T)$ uses S as argument to $\rightarrow$. These two cases imply the general case, since $\cap$ is nonexpansive and $\mathcal{R}^S(\mathtt{f}:R,\mathtt{m}:T,\ldots) = \mathcal{R}^S(\mathtt{f}:R)\cap\mathcal{R}^S(\mathtt{m}:T)\cap\ldots$.

The claim for the other arguments, R, T, ..., is handled similarly, since they too occur only as arguments to $\rightarrow$. □

Proposition 5
For all object type expressions `A` and all ρ, $\langle\mathtt{A}\rangle^S_\rho$ is contractive in S.

Proof
As for many of the propositions below, the argument is by induction on the structure of the proof that `A` is an object type expression. That is, we treat the cases of object type expressions of the forms `Root`, $\mathtt{A}[\mathtt{f}:\mathtt{B},\mathtt{m}:\mathtt{C}]$, and `Mu(X)A`. In the last two cases we assume, as induction hypothesis, that the claim is true for `A`.

- For `Root`, $\langle\mathtt{Root}\rangle^S_\rho$ is constant and hence contractive.
- By definition, $\langle\mathtt{A}[\mathtt{f}:\mathtt{B},\mathtt{m}:\mathtt{C}]\rangle^S_\rho$ is $\langle\mathtt{A}\rangle^S_\rho\cap\mathcal{R}^S(\mathtt{f}:[\![\mathtt{B}]\!]_\rho,\mathtt{m}:[\![\mathtt{C}]\!]_\rho)$. We obtain that $\langle\mathtt{A}[\mathtt{f}:\mathtt{B},\mathtt{m}:\mathtt{C}]\rangle^S_\rho$ is contractive in S using the induction hypothesis, Proposition 4, and the nonexpansiveness of $\cap$.
- By definition, $\langle\mathtt{Mu(X)A}\rangle^S_\rho$ is $\langle\mathtt{A}\rangle^S_{\rho\{\mathtt{X}\leftarrow[\![\mathtt{Mu(X)A}]\!]_\rho\}}$, and the induction hypothesis applies immediately (but using $\rho\{\mathtt{X}\leftarrow[\![\mathtt{Mu(X)A}]\!]_\rho\}$ as environment).

□

Corollary 2
For all $\mathtt{f}$, R, $\mathtt{m}$, T, ..., all object type expressions `A`, and all ρ,

$$\langle\mathtt{A}\rangle^S_\rho\cap\mathcal{R}^S(\mathtt{f}:R,\mathtt{m}:T,\ldots)$$

has a unique fixpoint as a function of S. This fixpoint is included in $L\rightarrow D$.

Proof
Since $\cap$ is nonexpansive, Propositions 4 and 5 yield that $\langle\mathtt{A}\rangle^S_\rho\cap\mathcal{R}^S(\mathtt{f}:R,\mathtt{m}:T,\ldots)$ is contractive in S, and hence has a unique fixpoint. Moreover, Propositions 2 and 3 show that this function has range $L\rightarrow D$, and hence its fixpoint is in $L\rightarrow D$. □

Proposition 6
For all object type expressions `A` and all ρ, $\langle\mathtt{A}\rangle^S_{\rho\{\mathtt{X}\leftarrow T\}}$ is contractive in T. For all type expressions `A` and all ρ, $[\![\mathtt{A}]\!]_{\rho\{\mathtt{X}\leftarrow T\}}$ is nonexpansive in T.

Proof
The claims are proved together, with an induction over a derivation that $\vdash$ `A obj` or $\vdash$ `A`:

- For $\mathtt{A} = \mathtt{X}$, the first result is vacuous (since `X` is not an object type expression) and the second one obvious.
- For `A` a function type, the first result is vacuous and the second one obvious.
- For $\mathtt{A} = \mathtt{Root}$, both results are easy, since `Root` does not depend on `X`.
- For $\mathtt{A} = \mathtt{A}'[\mathtt{f} : \mathtt{B}, \mathtt{m} : \mathtt{C}]$, the first claim follows from the induction hypothesis and Proposition 4, since $\cap$ preserves contractiveness; the second claim follows from the first one since $\cup$, $\cap$, and μ all preserve contractiveness.
- For $\mathtt{A} = \mathtt{Mu(X')A'}$, the first claim follows from the induction hypothesis, since $\langle \mathtt{Mu(X')A'} \rangle^S_{\rho\{\mathtt{X}\leftarrow T\}} = \langle \mathtt{A'} \rangle^S_{\rho\{\mathtt{X}\leftarrow T\}\{\mathtt{X'}\leftarrow [\![\mathtt{Mu(X')A'}]\!]_\rho\}}$; the second claim follows from the first one, as in the case of $\mathtt{A} = \mathtt{A}'[\mathtt{f} : \mathtt{B}, \mathtt{m} : \mathtt{C}]$.

$\square$

Corollary 3
For all object type expressions `A` and all ρ,

$$\bigcup\{\mu(S)(\langle \mathtt{A} \rangle^S_{\rho\{\mathtt{X}\leftarrow T\}} \cap \mathcal{R}^S(\mathtt{fd} : Rd, \mathtt{me} : Te, \ldots)) \mid \mathtt{fd}, \mathtt{me}, \ldots \text{ distinct labels not in } \mathtt{A}\}$$

has a unique fixpoint as a function of T. This fixpoint is included in $L{\rightarrow}D$.

Proof
The first part follows from Proposition 6, since $\cup$, $\cap$, and μ all preserve contractiveness. The second part follows from Corollary 2. $\square$

Proposition 7
For all `S`, S, and ρ,

$$[\![\mathtt{Root}]\!]_\rho = \langle \mathtt{Root} \rangle^S_\rho = [\![\mathtt{Root\ \ Self=S}]\!]_\rho = (L{\rightarrow}D)$$

Proof
We have $\langle \mathtt{Root} \rangle^S_\rho = (L{\rightarrow}D)$ from the definitions, and hence

$$[\![\mathtt{Root\ \ Self=S}]\!]_\rho = (L{\rightarrow}D)$$

Now we can calculate the semantics of `Root`. It is given as a union, and one of the sets that participates in this union is $\mu(S)\langle \mathtt{Root} \rangle^S_\rho$. Since $\langle \mathtt{Root} \rangle^S_\rho$ is identically $L{\rightarrow}D$, its fixpoint is $L{\rightarrow}D$. The other sets in the union are included in $L{\rightarrow}D$, by Corollary 2, so it follows that $[\![\mathtt{Root}]\!]_\rho = (L{\rightarrow}D)$. $\square$

Proposition 8
For all `f`, R, `m`, T, $\ldots$, $\mathcal{R}^S(\mathtt{f} : R, \mathtt{m} : T, \ldots)$ is antimonotonic in S.

Proof
This follows immediately from the antimonotonicity of $\rightarrow$ in its first argument, since $\mathcal{R}^S(\mathtt{f} : R, \mathtt{m} : T, \ldots)$ uses S as first argument to $\rightarrow$ in otherwise monotonic contexts. $\square$

Proposition 9

For all object type expressions $\mathtt{A}$ and all ρ, $\langle\mathtt{A}\rangle^S_\rho$ is antimonotonic in S.

Proof

This proof is almost identical to that of Proposition 5. $\square$

Note that Proposition 9 would be false if $\mathtt{A}$ was somehow allowed to refer to S. Some object-oriented languages provide constructs that support analogous references to the "Self" type. The use of a bounded intersection ("bounded quantification") might be of help in recovering from this problem. If S' is a new variable, the function $\cap_{S\subseteq S'}\langle\mathtt{A}\rangle^S_\rho$ is guaranteed to be antimonotonic in S'; it coincides with $\langle\mathtt{A}\rangle^S_\rho$ for the language we treat. The viability of this solution may deserve investigation.

On recursive types

Proposition 10

For all object type expressions $\mathtt{A}$ and all ρ, $[\![\mathtt{Mu(X)A}]\!]_\rho = [\![\mathtt{A}]\!]_{\rho\{\mathtt{X}\leftarrow[\![\mathtt{Mu(X)A}]\!]_\rho\}}$.

Proof

By definition, $[\![\mathtt{Mu(X)A}]\!]_\rho$ equals $\mu(T)[\![\mathtt{A}]\!]_{\rho\{\mathtt{X}\leftarrow T\}}$. In turn, $\mu(T)[\![\mathtt{A}]\!]_{\rho\{\mathtt{X}\leftarrow T\}}$ equals $[\![\mathtt{A}]\!]_{\rho\{\mathtt{X}\leftarrow[\![\mathtt{Mu(X)A}]\!]_\rho\}}$ by unfolding. $\square$

Proposition 11

If $\mathtt{A}$ and $\mathtt{B}$ are type expressions with the same infinite unfolding then $[\![\mathtt{A}]\!]_\rho = [\![\mathtt{B}]\!]_\rho$ for all ρ.

Proof

Proposition 10 shows the soundness of finite unfolding. The soundness of infinite unfolding follows because the semantics of a recursive type is the limit of the semantics of its finite unfoldings. (In a finite unfolding up to depth n, we "plug" with `Root` any branches that would go beyond depth n.) In turn, this limit property holds because $[\![\mathtt{D[f:X, m:Y]}]\!]$ is contractive in the interpretation of $\mathtt{X}$ and $\mathtt{Y}$, by Proposition 4, and because recursion can go only through the types of fields and methods in object types. $\square$

On subtyping

Proposition 12

For all object type expressions $\mathtt{A[f:B, m:C]}$ and all ρ,

$$[\![\mathtt{A[f:B, m:C]}]\!]_\rho \subseteq [\![\mathtt{A}]\!]_\rho$$

Proof

$$
\begin{aligned}
&[\![\mathtt{A[f:B,m:C]}]\!]_\rho \\
&\quad = \bigcup\{\mu(S)(\langle\mathtt{A[f:B,m:C]}\rangle^S_\rho \cap \mathcal{R}^S(\mathtt{fd}: Rd, \mathtt{me}: Te, \ldots)) \mid \\
&\qquad \mathtt{fd}, \mathtt{me}, \ldots \text{ distinct labels not in } \mathtt{A[f:B,m:C]}\} \\
&\quad = \bigcup\{\mu(S)(\langle\mathtt{A}\rangle^S_\rho \cap \mathcal{R}^S(\mathtt{f}: [\![\mathtt{B}]\!]_\rho, \mathtt{m}: [\![\mathtt{C}]\!]_\rho) \cap \mathcal{R}^S(\mathtt{fd}: Rd, \mathtt{me}: Te, \ldots)) \mid \\
&\qquad \mathtt{fd}, \mathtt{me}, \ldots \text{ distinct labels not in } \mathtt{A}, \text{ not } \mathtt{f}, \mathtt{m}\} \\
&\quad = \bigcup\{\mu(S)(\langle\mathtt{A}\rangle^S_\rho \cap \mathcal{R}^S(\mathtt{f}: [\![\mathtt{B}]\!]_\rho, \mathtt{m}: [\![\mathtt{C}]\!]_\rho, \mathtt{fd}: Rd, \mathtt{me}: Te, \ldots)) \mid \\
&\qquad \mathtt{fd}, \mathtt{me}, \ldots \text{ distinct labels not in } \mathtt{A}, \text{ not } \mathtt{f}, \mathtt{m}\} \\
&\quad \subseteq \bigcup\{\mu(S)(\langle\mathtt{A}\rangle^S_\rho \cap \mathcal{R}^S(\mathtt{fd}: Rd, \mathtt{me}: Te, \ldots)) \mid \\
&\qquad \mathtt{fd}, \mathtt{me}, \ldots \text{ distinct labels not in } \mathtt{A}\} \\
&\quad = [\![\mathtt{A}]\!]_\rho
\end{aligned}
$$

The inclusion step depends on the facts that `f` and `m` do not occur in `A` and that $[\![\mathtt{B}]\!]_\rho, [\![\mathtt{C}]\!]_\rho \in \mathbf{Idl}$. □

On extension and assignment

Proposition 13
If `A` is an object type expression with `no f,m in A`, ρ an environment, and $[\![\mathtt{a}]\!]_\rho \in \langle\mathtt{A}\rangle^S_\rho$, $[\![\mathtt{a}]\!]_\rho(\mathtt{f}) = *$, $[\![\mathtt{a}]\!]_\rho(\mathtt{m}) = *$, $[\![\mathtt{b}]\!]_\rho \neq *$, $[\![\mathtt{c}]\!]_\rho \neq *$, then $[\![\mathtt{a[f=b,m=c]}]\!]_\rho \in \langle\mathtt{A}\rangle^S_\rho$.

Proof
The argument is by induction on the structure of the proof that `A` is an object type expression:

- For `Root`, we use Proposition 3: if $[\![\mathtt{a}]\!]_\rho \in \langle\mathtt{Root}\rangle^S_\rho$ then $[\![\mathtt{a}]\!]_\rho \in (L{\to}D)$, and hence $[\![\mathtt{a[f=b,m=c]}]\!]_\rho \in \langle\mathtt{Root}\rangle^S_\rho$.
- Consider an object type expression of the form $\mathtt{A[fd:Bd,me:Ce]}$, with `fd` and `me` distinct from `f` and `m`. If $[\![\mathtt{a}]\!]_\rho \in \langle\mathtt{A[fd:Bd,me:Ce]}\rangle^S_\rho$ then $[\![\mathtt{a}]\!]_\rho \in \langle\mathtt{A}\rangle^S_\rho$, and then $[\![\mathtt{a[f=b,m=c]}]\!]_\rho \in \langle\mathtt{A}\rangle^S_\rho$ by $[\![\mathtt{a}]\!]_\rho(\mathtt{f}) = *$, $[\![\mathtt{a}]\!]_\rho(\mathtt{m}) = *$, $[\![\mathtt{b}]\!]_\rho \neq *$, $[\![\mathtt{c}]\!]_\rho \neq *$, and the induction hypothesis. In addition, $[\![\mathtt{a}]\!]_\rho \in \mathcal{R}^S(\mathtt{fd}: [\![\mathtt{Bd}]\!]_\rho, \mathtt{me}: [\![\mathtt{Ce}]\!]_\rho)$, and hence $[\![\mathtt{a}]\!]_\rho \in (L{\to}D)$; since `fd` and `me` are distinct from `f` and `m`, and $[\![\mathtt{a}]\!]_\rho(\mathtt{f}) = *$, $[\![\mathtt{a}]\!]_\rho(\mathtt{m}) = *$, $[\![\mathtt{b}]\!]_\rho \neq *$ and $[\![\mathtt{c}]\!]_\rho \neq *$, we get $[\![\mathtt{a[f=b,m=c]}]\!]_\rho \in \mathcal{R}^S(\mathtt{fd}: [\![\mathtt{Bd}]\!]_\rho, \mathtt{me}: [\![\mathtt{Ce}]\!]_\rho)$. Therefore, if $[\![\mathtt{a}]\!]_\rho \in \langle\mathtt{A[fd:Bd,me:Ce]}\rangle^S_\rho$ then $[\![\mathtt{a[f=b,m=c]}]\!]_\rho \in \langle\mathtt{A[fd:Bd,me:Ce]}\rangle^S_\rho$.
- The case of object type expressions of the form `Mu(X)A` is handled by unfolding the definition of $\langle\mathtt{Mu(X)A}\rangle^S_\rho$ and invoking the induction hypothesis with the environment $\rho\{\mathtt{X} \leftarrow [\![\mathtt{Mu(X)A}]\!]_\rho\}$.

(In this proof, as in many others below, the case of expressions that denote $\perp$ is rather trivial, since ideals are required to contain $\perp$ by definition; we tend to ignore this case.) □

Proposition 14
For all R, T, and S:

- If $o \in \mathcal{R}^S(\mathtt{f} : R)$ then $o(\mathtt{f}) \neq *$.
- If $o \in \mathcal{R}^S(\mathtt{m} : T)$ then $o(\mathtt{m}) \neq *$.

Proof
Since $R \in \mathbf{Idl}$, it cannot contain $*$. This settles the first claim. The argument for the second claim is almost identical, with $S{\rightarrow}T$ instead of R. □

Proposition 15
For all object type expressions $\mathtt{A}$, all S, and all ρ:

- If $[\![\mathtt{a}]\!]_\rho(\mathtt{f}) \neq *$ and $[\![\mathtt{b}]\!]_\rho \neq *$ then:
  - If $[\![\mathtt{a}]\!]_\rho \in \langle\mathtt{A}\rangle^S_\rho$ and $\mathtt{no\ f\ in\ A}$ then $[\![\mathtt{a.f} := \mathtt{b}]\!]_\rho \in \langle\mathtt{A}\rangle^S_\rho$.
  - If $[\![\mathtt{a}]\!]_\rho \in \mathcal{R}^S(\mathtt{fd} : Rd, \mathtt{me} : Te, \ldots)$ and $\mathtt{f}$ is not among $\mathtt{fd}$, $\mathtt{me}$, …then $[\![\mathtt{a.f} := \mathtt{b}]\!]_\rho \in \mathcal{R}^S(\mathtt{fd} : Rd, \mathtt{me} : Te, \ldots)$.
- If $[\![\mathtt{a}]\!]_\rho(\mathtt{m}) \neq *$ and $[\![\mathtt{c}]\!]_\rho \neq *$ then:
  - If $[\![\mathtt{a}]\!]_\rho \in \langle\mathtt{A}\rangle^S_\rho$ and $\mathtt{no\ m\ in\ A}$ then $[\![\mathtt{a.m} := \mathtt{c}]\!]_\rho \in \langle\mathtt{A}\rangle^S_\rho$.
  - If $[\![\mathtt{a}]\!]_\rho \in \mathcal{R}^S(\mathtt{fd} : Rd, \mathtt{me} : Te, \ldots)$ and $\mathtt{m}$ is not among $\mathtt{fd}$, $\mathtt{me}$, …, then $[\![\mathtt{a.m} := \mathtt{c}]\!]_\rho \in \mathcal{R}^S(\mathtt{fd} : Rd, \mathtt{me} : Te, \ldots)$.

Proof
For fields and for methods, the first claim is proved by induction on the structure of the proof that $\mathtt{A}$ is an object type expression, with a proof similar to that for Proposition 13; the second claim is proved directly from the definitions. □

On reading and invocation

We define an operator that reflects the self-application present in the semantics of object type expressions:

$$\lceil\mathtt{A}\rceil_\rho = \{o \in (L{\rightarrow}D) \mid o \in \langle\mathtt{A}\rangle^{\mathcal{C}\{o\}}_\rho\}$$

where $\mathcal{C}\{o\}$ is the least ideal containing o, that is, $\{v \mid v \sqsubseteq o\}$.

Proposition 16
For all object type expressions $\mathtt{A}$ and all ρ:

- If $\mathtt{f} : \mathtt{B\ in\ A}$ and $o \in \lceil\mathtt{A}\rceil_\rho$ then $o(\mathtt{f}) \in [\![\mathtt{B}]\!]_\rho$.
- If $\mathtt{m} : \mathtt{C\ in\ A}$ and $o \in \lceil\mathtt{A}\rceil_\rho$ then $o(\mathtt{m}) \in (D{\rightarrow}D)$ and $o(\mathtt{m})(o) \in [\![\mathtt{C}]\!]_\rho$.

Proof
We obtain the first result by induction on the structure of the proof that $\mathtt{A}$ is an object type:

- For $\mathtt{Root}$ the proof is vacuous, as $\mathtt{Root}$ has no fields or methods.
- For $\mathtt{A}[\mathtt{f} : \mathtt{B}, \mathtt{m}' : \mathtt{C}]$, the proof is immediate from the definitions.
- For $\mathtt{A}[\mathtt{f}' : \mathtt{B}', \mathtt{m}' : \mathtt{C}]$ with $\mathtt{f} \neq \mathtt{f}'$ the proof follows from the induction hypothesis.
- Recursive object type expressions are handled by unfolding.

The claim for methods is proved similarly. □

Proposition 17
For all object type expressions $\mathtt{A}$ and all ρ,

$$\mu(S)\langle\mathtt{A}\rangle_\rho^S \subseteq \lceil\mathtt{A}\rceil_\rho$$

Proof
The fixpoint considered exists, by Proposition 5. Now we argue by induction on the structure of the proof that $\mathtt{A}$ is an object type:

- If $\mathtt{A}$ is $\mathtt{Root}$, then the result follows from $\lceil\mathtt{Root}\rceil_\rho = (L{\rightarrow}D)$, which in turn follows from Proposition 7.
- Assume that $\mathtt{A}$ is $\mathtt{A}'[\mathtt{f}:\mathtt{B},\mathtt{m}:\mathtt{C}]$. Let $T = \mu(S)\langle\mathtt{A}\rangle_\rho^S$, and let v be an element of T. By unfolding, such a v is also in $\langle\mathtt{A}\rangle_\rho^T$. The definitions yield $v \in \langle\mathtt{A}'\rangle_\rho^T$. By Proposition 9, we also have $v \in \langle\mathtt{A}'\rangle_\rho^{\mathcal{C}\{v\}}$, since $\mathcal{C}\{v\} \subseteq T$. In addition, the definitions also give $v(\mathtt{f}) \in [\![\mathtt{B}]\!]_\rho$, and $v(\mathtt{m})(u) \in [\![\mathtt{C}]\!]_\rho$ for every $u \in T$. In particular, for $u = v$, we get $v(\mathtt{m})(v) \in [\![\mathtt{C}]\!]_\rho$. The properties of ideals yield that $v(\mathtt{m})(v') \in [\![\mathtt{C}]\!]_\rho$ for every $v' \sqsubseteq v$. Combining these results with $v \in \langle\mathtt{A}'\rangle_\rho^{\mathcal{C}\{v\}}$, we obtain the desired conclusion from the definitions.
- Assume that $\mathtt{A}$ is $\mathtt{Mu(X)A}'$. We have:

  $$\langle\mathtt{Mu(X)A}'\rangle_\rho^S = \langle\mathtt{A}'\rangle_{\rho\{\mathtt{X}\leftarrow[\![\mathtt{Mu(X)A}']\!]_\rho\}}^S$$

  and

  $$\lceil\mathtt{Mu(X)A}'\rceil_\rho = \lceil\mathtt{A}'\rceil_{\rho\{\mathtt{X}\leftarrow[\![\mathtt{Mu(X)A}']\!]_\rho\}}$$

  The result then follows from the induction hypothesis (used with the environment $\rho\{\mathtt{X}\leftarrow[\![\mathtt{Mu(X)A}']\!]_\rho\}$).

□

Main results

We say that ρ and $\mathtt{E}$ are consistent (and write $\rho \models \mathtt{E}$) if whenever $\mathtt{x}:\mathtt{A}$ occurs in $\mathtt{E}$ then $\rho(\mathtt{x}) \in [\![\mathtt{A}]\!]_\rho$.

Theorem 3
Assume that $\rho \models \mathtt{E}$. Then:

- If $\mathtt{E} \vdash \mathtt{A}$ then $[\![\mathtt{A}]\!]_\rho \in \mathbf{Idl}$.
- If $\mathtt{E} \vdash \mathtt{A}\ \mathtt{obj}$ then $[\![\mathtt{A}]\!]_\rho \in \mathbf{Idl}$ and $[\![\mathtt{A}]\!]_\rho \subseteq (L{\rightarrow}D)$.
- If $\mathtt{E} \vdash \mathtt{A} = \mathtt{B}$ then $[\![\mathtt{A}]\!]_\rho = [\![\mathtt{B}]\!]_\rho$.
- If $\mathtt{E} \vdash \mathtt{A} \leq \mathtt{B}$ then $[\![\mathtt{A}]\!]_\rho \subseteq [\![\mathtt{B}]\!]_\rho$.
- If $\mathtt{E} \vdash \mathtt{a}:\mathtt{A}$ then $[\![\mathtt{a}]\!]_\rho \in [\![\mathtt{A}]\!]_\rho$.
- If $\mathtt{E} \vdash \mathtt{a}:\mathtt{A}\ \ \mathtt{Self{=}S}$ then $[\![\mathtt{a}]\!]_\rho \in [\![\mathtt{A}\ \ \mathtt{Self{=}S}]\!]_\rho$.

Proof
The first claim follows immediately from the definitions. The second claim, discussed above, is a consequence of Corollary 2. The third claim follows from Proposition 11, which justifies the rules for equality of recursive object types. The fourth claim

follows from Proposition 12, which justifies the subtyping rule for object types. (The hypotheses of that rule imply that $\mathtt{A[f:B,m:C]}$ is an object type expression, so Proposition 12 is applicable.) It remains to check the soundness of the rules for typechecking; we discuss those related to objects.

- For
$$\frac{\mathtt{E} \vdash \mathtt{a:A\ \ Self{=}A}}{\mathtt{E} \vdash \mathtt{a:A}}$$
The assumption yields that $\mathtt{A}$ is an object type expression, by Proposition 1. Therefore, $[\![\mathtt{A}]\!]_\rho$ is defined as a union of fixpoints, one of which is $\mu(S)\langle\mathtt{A}\rangle^S_\rho$. This set equals $\langle\mathtt{A}\rangle^{\mu(S)\langle\mathtt{A}\rangle^S_\rho}_\rho$. Since $[\![\mathtt{A}]\!]_\rho \supseteq \mu(S)\langle\mathtt{A}\rangle^S_\rho$, Proposition 9 implies that $\langle\mathtt{A}\rangle^{\mu(S)\langle\mathtt{A}\rangle^S_\rho}_\rho \supseteq \langle\mathtt{A}\rangle^{[\![\mathtt{A}]\!]_\rho}_\rho$. In short, we obtain:
$$\begin{array}{rcl}[\![\mathtt{A}]\!]_\rho & \supseteq & \mu(S)\langle\mathtt{A}\rangle^S_\rho \\ & = & \langle\mathtt{A}\rangle^{\mu(S)\langle\mathtt{A}\rangle^S_\rho}_\rho \\ & \supseteq & \langle\mathtt{A}\rangle^{[\![\mathtt{A}]\!]_\rho}_\rho \\ & = & [\![\mathtt{A\ \ Self{=}A}]\!]_\rho\end{array}$$
- For
$$\frac{\mathtt{E} \vdash \mathtt{S}}{\mathtt{E} \vdash \mathtt{nil:Root\ \ Self{=}S}}$$
Since $[\![\mathtt{nil}]\!]_\rho \in [\![\mathtt{Root}]\!]_\rho$, the soundness of this rule follows from Proposition 7, which says that $[\![\mathtt{Root}]\!]_\rho = [\![\mathtt{Root\ \ Self{=}S}]\!]_\rho$ for any $\mathtt{S}$.
- For
$$\frac{\mathtt{E} \vdash \mathtt{a:A\ \ Self{=}S} \qquad \mathtt{E} \vdash \mathtt{b:B} \qquad \mathtt{E} \vdash \mathtt{c:D{\to}C} \qquad \mathtt{E} \vdash \mathtt{S{\leq}D} \qquad \mathtt{no\ f,m\ in\ A}}{\mathtt{E} \vdash \mathtt{a[f=b,m=c]:A[f:B,m:C]\ \ Self{=}S}}$$
By assumption, we have that $[\![\mathtt{a}]\!]_\rho \in [\![\mathtt{A\ \ Self{=}S}]\!]_\rho$, $[\![\mathtt{b}]\!]_\rho \neq *$, and $[\![\mathtt{c}]\!]_\rho \neq *$. Moreover, Proposition 1 guarantees that $\mathtt{A}$ is an object type expression and gives the form of $\mathtt{a}$, and from this form it follows that $[\![\mathtt{a}]\!]_\rho(\mathtt{f}) = *$ and $[\![\mathtt{a}]\!]_\rho(\mathtt{m}) = *$. Proposition 13 implies that $[\![\mathtt{a[f=b,m=c]}]\!]_\rho \in [\![\mathtt{A\ \ Self{=}S}]\!]_\rho$. The assumptions also yield $[\![\mathtt{b}]\!]_\rho \in [\![\mathtt{B}]\!]_\rho$ and $[\![\mathtt{c}]\!]_\rho \in [\![\mathtt{S}]\!]_\rho \to [\![\mathtt{C}]\!]_\rho$, and hence $[\![\mathtt{a[f=b,m=c]}]\!]_\rho \in \mathcal{R}^{[\![\mathtt{S}]\!]_\rho}(\mathtt{f}:[\![\mathtt{B}]\!]_\rho, \mathtt{m}:[\![\mathtt{C}]\!]_\rho)$.
Now it follows from the definition of $[\![\mathtt{A[f:B,m:C]\ \ Self{=}S}]\!]_\rho$ as the intersection of $[\![\mathtt{A\ \ Self{=}S}]\!]_\rho$ and $\mathcal{R}^{[\![\mathtt{S}]\!]_\rho}(\mathtt{f}:[\![\mathtt{B}]\!]_\rho, \mathtt{m}:[\![\mathtt{C}]\!]_\rho)$ that
$$[\![\mathtt{a[f=b,m=c]}]\!]_\rho \in [\![\mathtt{A[f:B,m:C]\ \ Self{=}S}]\!]_\rho$$
- For
$$\frac{\mathtt{E} \vdash \mathtt{a:A} \qquad \mathtt{E} \vdash \mathtt{A\ obj} \qquad \mathtt{f:B\ in\ A}}{\mathtt{E} \vdash \mathtt{a.f:B}}$$
The assumption that $\mathtt{E} \vdash \mathtt{a:A}$ means that:
$$\begin{array}{rcl}[\![\mathtt{a}]\!]_\rho & \in & \bigcup\{\mu(S)(\langle\mathtt{A}\rangle^S_\rho \cap \mathcal{R}^S(\mathtt{fd}:Rd, \mathtt{me}:Te, \ldots)) \mid \\ & & \quad \mathtt{fd}, \mathtt{me}, \ldots \text{ distinct labels not in } \mathtt{A}\}\end{array}$$

For $[\![\mathtt{a}]\!]_\rho$ finite, it follows that $[\![\mathtt{a}]\!]_\rho$ is in one of the sets that participate in the union, $\mu(S)(\langle\mathtt{A}\rangle^S_\rho \cap \mathcal{R}^S(\mathtt{fd}: Rd, \mathtt{me}: Te, \ldots))$. (Infinite elements are handled by continuity.) Proposition 17 guarantees that $[\![\mathtt{a}]\!]_\rho$ is in $\lceil \mathtt{A}[\mathtt{fd}: \mathtt{Rd}, \mathtt{me}: \mathtt{Te}, \ldots] \rceil_{\rho'}$, where $\mathtt{Rd}$, $\mathtt{Te}$, … are new type variables and ρ' extends ρ to map them to Rd, Te, …. We also obtain that $[\![\mathtt{a}]\!]_\rho \in (L \to D)$.

Now, $\mathtt{A}[\mathtt{fd}: \mathtt{Rd}, \mathtt{me}: \mathtt{Te}, \ldots]$ is an extension of $\mathtt{A}$, and hence $\mathtt{f}: \mathtt{B}$ `in` $\mathtt{A}$ implies $\mathtt{f}: \mathtt{B}$ `in` $\mathtt{A}[\mathtt{fd}: \mathtt{Rd}, \mathtt{me}: \mathtt{Te}, \ldots]$. Since $[\![\mathtt{a}]\!]_\rho \in (L \to D)$, $[\![\mathtt{a.f}]\!]_\rho = [\![\mathtt{a}]\!]_\rho(\mathtt{f})$, and Proposition 16 yields that $[\![\mathtt{a.f}]\!]_\rho \in [\![\mathtt{B}]\!]_\rho$.

- For

$$\frac{\mathtt{E} \vdash \mathtt{a}: \mathtt{A} \qquad \mathtt{E} \vdash \mathtt{A}\ \mathtt{obj} \qquad \mathtt{m}: \mathtt{C}\ \mathtt{in}\ \mathtt{A}}{\mathtt{E} \vdash \mathtt{a.m}: \mathtt{C}}$$

The soundness argument for this rule resembles the previous one, using Propositions 16 and 17. At the end of the argument, we use an additional fact obtained from Proposition 16, that $[\![\mathtt{a}]\!]_\rho(\mathtt{m}) \in (D \to D)$; this is needed for guaranteeing that $[\![\mathtt{a.m}]\!]_\rho = [\![\mathtt{a}]\!]_\rho(\mathtt{m})([\![\mathtt{a}]\!]_\rho)$.

- For

$$\frac{\mathtt{E} \vdash \mathtt{a}: \mathtt{A} \qquad \mathtt{E} \vdash \mathtt{b}: \mathtt{B} \qquad \mathtt{f}: \mathtt{B}\ \mathtt{in}\ \mathtt{A}}{\mathtt{E} \vdash \mathtt{a.f} := \mathtt{b}: \mathtt{A}}$$

Since $\mathtt{f}: \mathtt{B}$ `in` $\mathtt{A}$, $\mathtt{A}$ must be an object type expression, and we can assume it is of the form $\mathtt{A}'[\mathtt{f}: \mathtt{B}, \mathtt{m}: \mathtt{C}]$, all other cases being similar to this one. Assume further that $[\![\mathtt{a}]\!]_\rho$ is finite. (Infinite elements are handled by continuity.)

If $[\![\mathtt{a}]\!]_\rho \in [\![\mathtt{A}]\!]_\rho$ then $[\![\mathtt{a}]\!]_\rho$ is in some ideal $T \subseteq [\![\mathtt{A}]\!]_\rho$ of the form

$$\mu(S)(\langle\mathtt{A}\rangle^S_\rho \cap \mathcal{R}^S(\mathtt{fd}: Rd, \mathtt{me}: Te, \ldots))$$

which equals

$$\mu(S)(\langle\mathtt{A}'\rangle^S_\rho \cap \mathcal{R}^S(\mathtt{f}: [\![\mathtt{B}]\!]_\rho, \mathtt{m}: [\![\mathtt{C}]\!]_\rho) \cap \mathcal{R}^S(\mathtt{fd}: Rd, \mathtt{me}: Te, \ldots))$$

and, by unfolding,

$$\langle\mathtt{A}'\rangle^T_\rho \cap \mathcal{R}^T(\mathtt{f}: [\![\mathtt{B}]\!]_\rho, \mathtt{m}: [\![\mathtt{C}]\!]_\rho) \cap \mathcal{R}^T(\mathtt{fd}: Rd, \mathtt{me}: Te, \ldots)$$

So $[\![\mathtt{a}]\!]_\rho \in \mathcal{R}^T(\mathtt{f}: [\![\mathtt{B}]\!]_\rho)$, and, by Proposition 14, $[\![\mathtt{a}]\!]_\rho(\mathtt{f}) \neq *$. If in addition $[\![\mathtt{b}]\!]_\rho \in [\![\mathtt{B}]\!]_\rho$, then $[\![\mathtt{a.f} := \mathtt{b}]\!]_\rho \in \mathcal{R}^T(\mathtt{f}: [\![\mathtt{B}]\!]_\rho)$. Since $\mathtt{A}'[\mathtt{f}: \mathtt{B}, \mathtt{m}: \mathtt{C}]$ is an object type expression, we have `no f in` $\mathtt{A}'$, and $\mathtt{f}$ differs from $\mathtt{fd}$, $\mathtt{me}$, …, so Proposition 15 applies, and yields

$$[\![\mathtt{a.f} := \mathtt{b}]\!]_\rho \in \langle\mathtt{A}'\rangle^T_\rho \cap \mathcal{R}^T(\mathtt{f}: [\![\mathtt{B}]\!]_\rho, \mathtt{m}: [\![\mathtt{C}]\!]_\rho) \cap \mathcal{R}^T(\mathtt{fd}: Rd, \mathtt{me}: Te, \ldots)$$

that is, $[\![\mathtt{a.f} := \mathtt{b}]\!]_\rho \in T$, hence $[\![\mathtt{a.f} := \mathtt{b}]\!]_\rho \in [\![\mathtt{A}]\!]_\rho$.

- For

$$\frac{\mathtt{E} \vdash \mathtt{a}: \mathtt{A} \qquad \mathtt{E} \vdash \mathtt{g}: \mathtt{D} \to \mathtt{C} \qquad \mathtt{E} \vdash \mathtt{A} \leq \mathtt{D} \qquad \mathtt{m}: \mathtt{C}\ \mathtt{in}\ \mathtt{A}}{\mathtt{E} \vdash \mathtt{a.m} := \mathtt{g}: \mathtt{A}}$$

The proof is similar to the previous one. The only difference is that here we use $[\![\mathtt{g}]\!]_\rho \in [\![\mathtt{A}]\!]_\rho \to [\![\mathtt{C}]\!]_\rho$ and $[\![\mathtt{A}]\!]_\rho \supseteq T$ in order to derive that $[\![\mathtt{a.f} := \mathtt{g}]\!]_\rho \in \mathcal{R}^T(\mathtt{m}: [\![\mathtt{C}]\!]_\rho)$.

The treatment of the rules not related to objects is standard. $\square$

Corollary 4
If $\rho \models \mathtt{E}$ and $\mathtt{E} \vdash \mathtt{a} : \mathtt{A}$ then $[\![\mathtt{a}]\!]_\rho \neq *$.

Proof
If $\rho \models \mathtt{E}$ and $\mathtt{E} \vdash \mathtt{a} : \mathtt{A}$ then $[\![\mathtt{a}]\!]_\rho \in [\![\mathtt{A}]\!]_\rho$ by Theorem 3. Moreover, $\mathtt{E} \vdash \mathtt{a} : \mathtt{A}$ yields $\mathtt{E} \vdash \mathtt{A}$, and so $[\![\mathtt{A}]\!]_\rho \in \mathbf{Idl}$ by Theorem 3. Since $* \notin [\![\mathtt{A}]\!]_\rho$, we obtain $[\![\mathtt{a}]\!]_\rho \neq *$. $\square$

Corollary 5
If $\mathtt{a} \Rightarrow \mathtt{a}'$ and $\vdash \mathtt{a} : \mathtt{A}$ then $\mathtt{a}'$ is not `wrong`.

Proof
Any ρ is consistent with the empty environment, so if $\vdash \mathtt{a} : \mathtt{A}$ then $[\![\mathtt{a}]\!]_\rho \neq *$, by Corollary 4. In addition, if $\mathtt{a} \Rightarrow \mathtt{a}'$ then $[\![\mathtt{a}]\!] = [\![\mathtt{a}']\!]$, by Theorem 2. Hence $[\![\mathtt{a}']\!]_\rho \neq *$, and $\mathtt{a}'$ is not `wrong`. $\square$

4.3 Reasoning about programs

The denotational semantics can also serve in validating rules for reasoning about programs. We only start the explorations of such rules, by giving two simple examples.

- For assignment to fields, we have an inequational rule. The relation $\sqsubseteq$ is the evident syntactic representation of the domain order, as in Scott's LCF.

$$\frac{\mathtt{E} \vdash \mathtt{a} : \mathtt{A} \qquad \mathtt{E} \vdash \mathtt{b} : \mathtt{B} \qquad \mathtt{f} : \mathtt{B}\ \mathtt{in}\ \mathtt{A}}{\mathtt{E} \vdash (\mathtt{a.f} := \mathtt{b}).\mathtt{f} \sqsubseteq \mathtt{b} : \mathtt{B}}$$

  In order to justify the rule, we observe that $(\mathtt{a.f} := \mathtt{b}).\mathtt{f}$ differs from $\mathtt{b}$ only when it denotes $\perp$ or $*$; moreover, the hypotheses exclude this last possibility. An analogue for this rule in an imperative setting might be that if $\mathtt{P}$ is a predicate, $\mathtt{P}(\mathtt{b})$ holds before the assignment $\mathtt{a.f} := \mathtt{b}$, and this assignment terminates, then $\mathtt{P}(\mathtt{a.f})$ holds afterwards.
- A similar inequational rule is sound for overriding:

$$\frac{\mathtt{E} \vdash \mathtt{a} : \mathtt{A} \qquad \mathtt{E} \vdash \mathtt{c} : \mathtt{D} \to \mathtt{C} \qquad \mathtt{E} \vdash \mathtt{A} \leq \mathtt{D} \qquad \mathtt{m} : \mathtt{C}\ \mathtt{in}\ \mathtt{A}}{\mathtt{E} \vdash (\mathtt{a.m} := \mathtt{c}).\mathtt{m} \sqsubseteq \mathtt{c}(\mathtt{a}) : \mathtt{C}}$$

A useful project would be to extend the denotational semantics to a larger fragment of Modula-3, and then prove the soundness of a verification system for that language. This project is appealing because Modula-3 was designed with formal methods in mind and there are active efforts in the specification and verification of Modula-3 programs (Cardelli and Nelson, 1993; Guttag and Horning, 1993).

4.4 A stronger semantics of types

The ideal semantics of Subsection 4.2 does not validate all reasonable rules. For example, we might expect that a function in $\mathtt{Root} \to \mathtt{Nat}$ be constant, but it need not be in the ideal semantics. A stronger semantics may be based on per models (e.g., (Amadio, 1991; Cardone, 1989; Abadi and Plotkin, 1990)) or, perhaps better, on parametric per models (e.g., (Bainbridge *et al.*, 1990)).

For the sake of simplicity, we do not use pers in the body of this paper. Here we sketch the modifications necessary for obtaining a per semantics, and then discuss the result. As Amadio and Cardone, we take a metric approach. Finding a per semantics along the lines of (Abadi and Plotkin, 1990) remains a challenge.

A complete uniform per is a symmetric, transitive, binary relation R on D with the properties:

- R is nonempty;
- if uRv then $(p_i(u))R(p_i(v))$ for all i;
- R is closed under limits of increasing sequences in the $\sqsubseteq$ order.

The distance between two pers is 2^{-r}, where r is the minimum rank where the two pers differ, and it is 0 if the two pers are equal.

The complete uniform pers that do not relate $*$ to any value provide suitable denotations for type expressions. The collection of all such complete uniform pers is **CUPer**.

The changes required in replacing **Idl** with **CUPer** are mostly local. Like ideals, complete uniform pers can be combined with intersection and (not as easily) with union. Furthermore, there is a suitable function-space operator:

$$R \to T = \{(\bot, \bot)\} \cup \{(f, g) \in (D \to D) \times (D \to D) \mid \text{if } xRy \text{ then } f(x)Tg(y)\}$$

and we can solve fixpoint equations. As for object types, we update the definitions of $\langle \mathtt{Root} \rangle_\rho^S$ and $\mathcal{R}^S(\mathtt{f} : R, \mathtt{m} : T, \ldots)$; they become:

$$\begin{array}{rcl} \langle \mathtt{Root} \rangle_\rho^S & = & (L \to D) \times (L \to D) \\ \mathcal{R}^S(\mathtt{f} : R, \mathtt{m} : T, \ldots) & = & \{(o, o') \in (L \to D) \times (L \to D) \mid \\ & & (o(\mathtt{f}), o'(\mathtt{f})) \in R \wedge (o(\mathtt{m}), o'(\mathtt{m})) \in (S \to T) \wedge \ldots\} \end{array}$$

Here, as in many other obvious places, we replace $L \to D$ with $(L \to D) \times (L \to D)$. With this change, the propositions up to Proposition 12 are proved as for ideals. To adapt Propositions 13–15, we interpret $v \in R$ as $(v, v) \in R$, for v a value and R a per. For Propositions 16 and 17, we note that the least complete uniform per containing o is $\mathcal{C}\{o\} = \{(o, o)\} \cup \bigcup_i \{(p_i(o), p_i(o))\}$. The main results follow.

The advantages of pers over ideals in the semantics of typed lambda calculi are well known (see for example (Gunter, 1992, p. 266)). Basically, pers validate the ξ rule, according to which if $\mathtt{b}$ and $\mathtt{b}'$ are equal as elements of $\mathtt{B}$ for all $\mathtt{x}$ in $\mathtt{A}$ then $\mathtt{fun(x : A)b}$ and $\mathtt{fun(x : A)b}'$ are equal as elements of $\mathtt{A} \to \mathtt{B}$. We benefit from this in Baby Modula-3, which is an extension of a typed lambda calculus. We also obtain new equalities of objects. For example, the functions in $\mathtt{Root} \to \mathtt{Nat}$ are constant. Further, if $\mathtt{A}$ is $\mathtt{Root[f : Nat, m : Nat]}$ and $\mathtt{c}$ and $\mathtt{c}'$ are equal as elements of $\mathtt{A} \to \mathtt{Nat}$ then $\mathtt{nil[f = 0, m = c]}$ and $\mathtt{nil[f = 0, m = c']}$ are equal as elements of $\mathtt{A}$.

The per semantics does not seem to validate all reasonable equations, however. Consider $\mathtt{A}' = \mathtt{Root[f : Nat, m : Nat][f' : Nat, m' : Nat]}$, a subtype of $\mathtt{A}$. The objects

$$\mathtt{nil[f = 0, m = fun(x : A)0]}$$

and

$$\mathtt{nil[f = 0, m = fun(x : A')(x.f')][f' = 0, m' = fun(x : A')(x.m')]}$$

are not equal as elements of `A`, although they behave identically in any context that treats them as elements of `A`.

5 Related work

In the last few years there have been diverse works on the foundations of object-oriented programming. Some focused on untyped languages, for example Cook's thesis (1989). We have mentioned the influential papers of Cardelli and Mitchell, which concern typed languages. Here we discuss other works on typed languages. They are very recent and ongoing, and they seem to be the first to present thorough soundness results. The exact relations between the approaches are not entirely clear at this point.

We can classify formal accounts of object-oriented languages along two dimensions, the language treated and the description method used:

1. The language treated. There are several main families of object-oriented language. In class-based languages, methods are attached to classes, which are used to generate objects; in delegation-based languages, methods are attached to individual objects. In particular, delegation-based languages may allow overriding methods in individual objects (like Baby Modula-3). Such a feature would be problematic in the class-based languages discussed below.
2. The description method used. Some of the accounts are based on syntactic translations into more or less traditional higher-order languages, such as System F enriched with subtyping, recursion, and records; when the target language chosen is sufficiently well understood, this yields a denotational semantics as a side-product. Other accounts give a direct denotational semantics.

Continuing his original work, now with Honsell and Fisher, Mitchell presents a delegation-based language (1993). The untyped version of this language and that of Baby Modula-3 are quite similar. The type systems seem incomparable: Mitchell et al. concentrate on inheritance, but do not provide a subtype relation. Their study is syntactic, and the main technical result is a subject-reduction theorem.

Bruce (1993) treats a class-based language called TOOPL. The language includes a rich object system. It does not allow explicit recursion; rather, some recursion is obtained through the class mechanisms. Bruce's technique draws on a fairly long line of previous papers, such as (Cook *et al.*, 1990). The method is essentially semantic, but parts of the constructions can be seen as translations into a language with recursion and F-bounded universal quantifiers. The result is rather complicated. However, it is possible that this complexity is intrinsic to the project of giving a semantics to TOOPL.

Another interesting approach is that of Pierce and Turner (1993). Again, they treat a class-based language. This language is more limited than Bruce's, and in particular it lacks binary methods. (Pierce and Turner have gone on to propose a new way to model binary methods (1992).) The semantics of the language is based on a translation, and it exploits abstract data types rather than polymorphic types and recursion.

Castagna, Ghelli, and Longo (1992; 1992) suggest a very different view of object-oriented programming languages. They present a core calculus, with classes, subtyping, and overloaded functions. It leads to an original treatment of constructs such as multiple dispatch in the CLOS style (Steele, 1990). (All the other papers discussed here deal only with single dispatch.)

Modula-3 is a rather traditional language, with no classes, and so is Baby Modula-3. We present a semantic definition, but not a translation into a standard typed lambda calculus. It is however possible that the semantic definition may lead to such a translation. In particular, the union operation in our semantics of object types may correspond to an existential quantifier. Other semantic constructs clearly correspond to record types, and in that our work continues that of Cardelli and Mitchell. We leave as an open problem the definition of a translation from Baby Modula-3 into a standard typed lambda calculus.

Thus, Baby Modula-3 differs significantly from the other languages used in formal studies, and the theory of objects presented relies on some new ideas and constructions. However, the various theories of objects seem compatible. A synthesis might be both viable and useful.

Acknowledgments

I would especially like to thank Luca Cardelli, with whom I originally wrote type rules for Baby Modula-3 and who helped me in some difficult choices. Luca Cardelli and Kim Bruce helped in clarifying the relation between this and previous work. Bill Kalsow provided useful comments on the presentation and explained delicate aspects of Modula-3. Bob Harper and Benjamin Pierce provided further comments on the presentation. Cynthia Hibbard suggested stylistic improvements. Finally, anonymous referees suggested many improvements both of contents and of form.

References

Abadi, M. and Plotkin, G. 1990. A per model of polymorphism and recursive types. In *Proceedings of the Fifth Annual Symposium on Logic In Computer Science Conference*, pages 355–365. IEEE Computer Society.

Amadio, R. and Cardelli, L. 1991. Subtyping recursive types. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 104–118. ACM.

Amadio, R. 1991. Recursion over realizability structures. *Information and Computation*, 91(1):55–85.

Bainbridge, E. S., Freyd, P. J., Scedrov, A. and Scott, P. J. 1990. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64. Corrigendum in (3)71, April 1990, page 431.

Bruce, K. 1993. Safe type checking in a statically-typed object-oriented programming language. In *Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages*, pages 285–298. ACM.

Cardelli, L. 1986. Amber. In Cousineau, G., Curien, P.-L. and Robinet, B., editors, *Combinators and Functional Programming Languages*. Lecture Notes in Computer Science No. 242. Springer-Verlag.

Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B. and Nelson, G. 1988. Modula-3 report. Research Report 31, Digital Equipment Corporation Systems Research Center.

Cardelli, L. 1992. Extensible records in a pure calculus of subtyping. In Gunter, C. and Mitchell, J. C., editors, *Theoretical Aspects of Object-oriented Programming: Types, Semantics and Language Design*. MIT Press, to appear. A preliminary version has appeared as SRC Research Report No. 81.

Cardelli, L. and Nelson, G. 1993. Structured command semantics. Draft.

Cardone, F. 1989. Relational semantics for recursive types and bounded quantification. In Ausiello, G., Dezani-Ciancaglini, M. and Ronchi Della Rocca, S., editors, *Automata, Languages and Programming*. Lecture Notes in Computer Science No. 372, pages 164–178. Springer-Verlag.

Castagna, G., Ghelli, G. and Longo, G. 1992. A calculus for overloaded functions with subtyping. Technical Report LIENS-92-4, Ecole Normale Supérieure.

Castagna, G. 1992. Strong typing in object-oriented paradigms. Technical Report LIENS-92-11, Ecole Normale Supérieure.

Cook, W. R. 1989. *A denotational semantics of inheritance*. PhD thesis, Brown University.

Cook, W. R., Hill, W. L. and Canning, P. S. 1990. Inheritance is not subtyping. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135. ACM.

Girard, J.-Y. 1972. *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII.

Gunter, C. 1992. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing Series. The MIT Press, Cambridge, Massachusetts.

Guttag, J. V. and Horning, J. J., editors. 1993. *Larch: Languages and Tools for Formal Specification*. Texts and monographs in computer science. Springer-Verlag.

MacQueen, D., Plotkin, G. and Sethi, R. 1986. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130.

Mitchell, J. C. 1990. Toward a typed foundation for method specialization and inheritance. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 109–124. ACM.

Mitchell, J. C., Honsell, F. and Fisher, K. 1993. A lambda calculus of objects and method specialization. In *Proceedings of the Eight IEEE Annual Symposium on Logic in Computer Science*, pages 26–38. IEEE Computer Society.

Nelson, G., editor. 1991. *Systems Programming in Modula-3*. Prentice Hall.

Pierce, B. C. and Turner, D. N. 1992. Statically typed multi-methods via partially abstract types. Draft.

Pierce, B. C. and Turner, D. N. 1993. Object-oriented programming without recursive types. In *Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages*, pages 299–312. ACM.

Plotkin, G. 1981. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark.

Steele, G. L. 1990. *Common Lisp: The Language*. Digital Press, Bedford, Massachusetts, second edition.

Wand, M. 1987. Complete type inference for simple objects. In *Proceedings of the Second Symposium on Logic in Computer Science*, pages 37–44. IEEE Computer Society. Corrigendum in *Proceedings of the Third Symposium on Logic in Computer Science*, page 132, 1988.