Abstract types and the dot notation

Luca Cardelli, Xavier Leroy

Abstract types and the dot notation

Luca Cardelli Xavier Leroy

Research report 56
Digital Equipment Corporation, Systems Research Center
March 10, 1990

Authors' abstract

We investigate the use of the dot notation in the context of abstract types. The dot notation—that is, $a.f$ referring to the operation f provided by the abstraction a—is used by programming languages such as Modula-2 and CLU. We compare this notation with the Mitchell-Plotkin approach, which draws a parallel between type abstraction and (weak) existential quantification in constructive logic. The basic operations on existentials coming from logic give new insights about the meaning of type abstraction, but differ completely from the more familiar dot notation. In this paper, we formalize simple calculi equipped with the dot notation, and relate them to a more classical calculus *à la* Mitchell and Plotkin. This work provides some theoretical foundations for the dot notation, and suggests some useful extensions.

Publication history

This report is based on a paper presented at the IFIP TC2 working conference on Programming Concepts and Methods, Tiberias, Israel, april 1990. (Proceedings published by North-Holland in 1990.)

Contents

1 Introduction

Type abstraction has emerged as one of the most important techniques for specifying and building large software systems [6, 4], since it provides fundamental typing support for modularization [14].

Abstract types (sometimes called opaque types) are therefore one of the necessary features of modern programming languages. However, for a long time their standing has been rather mysterious, and their type rules have been explained in ad-hoc and operational ways, making formal reasoning about abstract types difficult. For example, it is still commonly said that an opaque type is "different from any other type in the system when seen from outside the abstraction", or that a new abstract type is "created" whenever its description is evaluated. Such statements are both informal and arbitrary; clearly, a formal background is needed to define precisely what type abstraction means, derive sensible typechecking rules, and reason about programs using abstract types.

In an attempt to fill this need, Mitchell and Plotkin [13] made an important connection between type abstraction and the second-order existential quantifiers of logic. They proposed that an abstraction—that is, an abstract type A, together with operations $f, g, \ldots$ whose types $F, G, \ldots$ normally involve A—should be viewed as an "existential" statement. That is, there should exist a concrete type representation of A and an implementation of the operations $f, g, \ldots$ such that $f, g, \ldots$ have the types $F, G, \ldots$, respectively[1]. For instance, a package implementing complex numbers could be specified by the existential type shown in figure 1; two different implementations meeting this specification are shown in figure 2. An abstraction is therefore an assertion that adequate implementations exist; it provides a partial specification of such implementations. These existential statements might be false, in which case the specification of the abstract type should be seen as inconsistent.

The approach of Mitchell and Plotkin showed for the first time that the type rules for abstract types could be described non-operationally, by looking at the well-known rules of constructive logic from the standpoint of programming. It also provided the necessary formal framework for proving fundamental properties of abstract types, such as representation independence [15, 11].

The connection between this approach to type abstraction and the notion of type abstraction found in several modern programming languages is however not complete. These languages use a *dot notation*, such as $a.f$, to refer to an operation f provided by an abstraction a, or in other words, to the field named f of module a. The type theory approach provides an elimination construct that looks totally different, for the same purpose . We set out in this paper to investigate this difference in notation, explore

[1] This type-theoretical notion of "abstract types", should not be confused with the many-sorted algebra approach. It is both weaker, since it does not involve equations (which can however be added in a type-theoretical logic), and also stronger, because of higher-order functions.

$$
\begin{array}{l}
\texttt{type}\ \mathit{Complex} = \\
\quad \exists C.\ \mathit{make} \ : \ \texttt{Real} \to \texttt{Real} \to C \\
\qquad\ \ \mathit{re} \quad : \ C \to \texttt{Real} \\
\qquad\ \ \mathit{im} \quad : \ C \to \texttt{Real} \\
\qquad\ \ \mathit{mul} \ : \ C \to C \to C \quad \ldots
\end{array}
$$

Figure 1: A specification of a complex arithmetic package

$$
\begin{array}{l}
\texttt{val}\ \mathit{cartesian\_complex} : \mathit{Complex} = \\
\quad \langle\ C \ = \ \texttt{Real} \times \texttt{Real} \\
\quad\ \ \mathit{make} \ = \ \lambda x{:}\texttt{Real}.\, \lambda y{:}\texttt{Real}.\, (x, y) \\
\quad\ \ \mathit{re} \ = \ \lambda x{:}C.\, \texttt{first}(x) \\
\quad\ \ \mathit{im} \ = \ \lambda x{:}C.\, \texttt{second}(x) \\
\quad\ \ \mathit{mul} \ = \ \lambda x{:}C.\, \lambda y{:}C.\ (\texttt{first}(x).\texttt{first}(y) - \texttt{second}(x).\texttt{second}(y), \\
\qquad\qquad\qquad\qquad\ \ \texttt{first}(x).\texttt{second}(y) + \texttt{second}(x).\texttt{first}(y))\ \rangle \\
\texttt{val}\ \mathit{polar\_complex} : \mathit{Complex} = \\
\quad \langle\ C \ = \ \texttt{Real} \times \texttt{Real} \\
\quad\ \ \mathit{make} \ = \ \lambda x{:}\texttt{Real}.\, \lambda y{:}\texttt{Real}.\, (\sqrt{x^2 + y^2}, \texttt{atan2}(y, x)) \\
\quad\ \ \mathit{re} \ = \ \lambda x{:}C.\, \texttt{first}(x).\texttt{cos}(\texttt{second}(x)) \\
\quad\ \ \mathit{im} \ = \ \lambda x{:}C.\, \texttt{first}(x).\texttt{sin}(\texttt{second}(x)) \\
\quad\ \ \mathit{mul} \ = \ \lambda x{:}C.\, \lambda y{:}C.\, (\texttt{first}(x).\texttt{first}(y), \texttt{second}(x) + \texttt{second}(y))\ \rangle
\end{array}
$$

Figure 2: Two implementations of the complex arithmetic package

```
λcomplex : Complex.
  open complex as ⟨C, mk, re, im, mul⟩ in
    let square = λx:C. mul x x in re(square(mk 2 3))
```

Figure 3: Using a package with the open notation

```
λcomplex : Complex.
  let square = λx : complex.C. (complex.mul x x) in
    complex.re(square(complex.make 2 3))
```

Figure 4: Using a package with the dot notation

also whether there is a difference in expressive power, and try to relate both notations, in the hope of filling one of the remaining gaps between the theory and the practice of type abstraction.

The "logical" notation for abstraction uses a construct, which we call `open` here, corresponding to the logical rule for existential elimination. Namely, given a package p implementing an abstraction $A, f, g, \ldots$, we can open the package p, bind its components to variables $A', f', g', \ldots$, and use such variables in a usage scope (following `in`):

$$p : \exists A.\ f, g, \ldots$$
$$\texttt{open}\ p\ \texttt{as}\ A', f', g', \ldots\ \texttt{in}\ \ldots A' \ldots f' \ldots g' \ldots$$

The result of this expression is the result of the subexpression following `in`. Figure 3 shows how the complex arithmetic package above can be used with this notation. Additional examples using this notation can be found in Cardelli and Wegner [3].

It is easy to see why abstraction is enforced. Although A may be implemented in p as, say, `Int`, A' is a formal variable and cannot match any type except itself. This corresponds to the informal idea that abstract types are "different from any other type". There is also a crucial restriction that the type variable A' should not escape its scope: it must not appear free in the type of the result of `open` or in the types of global variables.

Existential types provide a fully satisfactory theoretical solution to the problem of modelling abstract types. However, the open notation is very clumsy for programming purposes; if a package is opened twice, the two abstract type variables thus introduced will not match. Hence one must find a sufficiently large usage scope around which to place the `open` construct without violating the typing rules; MacQueen [9] points out that the required usage scope may be so large that most benefits of abstraction are lost.

This difficulty motivated MacQueen to take a different approach to abstraction, using dependent types and general sums [9]. His approach is adequate for modules (when they are not first-class values, as in Standard ML [8]), but not for abstract types, since packages

must now necessarily stop being first-class values to preserve typing decidability [12]. In addition, general sums do not ensure abstraction by themselves; hence, Standard ML provides a separate abstraction construct.

Independently, programming languages providing facilities for type abstraction were developed, and most of them use a different notation. This programming notation uses a *dot* operator to select the components of a package, including selecting the abstract type when it is in a type context:

$$p : \exists A.\ f, g, \ldots$$
$$\ldots p.A \ldots p.f \ldots p.g \ldots$$

See figure 4 for a reformulation of the example above using this notation.

This dot notation has proved very convenient for actual programming, especially for programming in the large. However, it does not lend itself to formalization as readily as the open notation.

One difficulty with the dot notation is that the name A is bound and can be renamed: $\exists A.B = \exists A'.B\{A \leftarrow A'\}$. Hence what does $p.A$ really mean? Similarly, how are the names $f, g, \ldots$ handled? Programming languages seem to avoid this problem by rejecting α-conversion of existential variables. Here we circumvent the problem by using positions instead of names: $p.\texttt{Fst}$ refers to the type component of an abstraction, and $p.\texttt{snd}$ to the operation-set component.

Apart from this difficulty, the main problem is that we no longer have a precise notion of the usage scope of a package. If $p.A$ always matches (another occurrence of) $p.A$, why should it not match `Int`, or some other type?

Moreover, $p.A$ (or $p.\texttt{Fst}$) is now a type; hence we have introduced value expressions as subexpressions of type expressions. Should $p.A$ match $p'.A$ only if $p = p'$? Or if p and p' have the same normal form? Clearly some restriction should be imposed here to preserve typing decidability.

Finally, are packages first-class values? In Modula-2 for instance, modules are not first-class values. In this paper, we wish to avoid this additional stratification, and assume that the p in $p.A$ is a first-class value, so we can handle general abstract types and not just modules. Alternatively, we are assuming that our modules are first-class values and that multiple implementations of an interface can coexist [2]. But should we require p in $p.A$ to be a simple identifier x, as in most programming languages, or a sequence $x.y.z\ldots$? Or can we allow p to be an arbitrarily complex expression, provided that it has no side-effect, and that equality of two such expressions is decidable?

If we had a translation between some flavor of dot notation and the open notation, we could avoid answering directly all such hard questions about the dot notation. We should not be afraid of losing expressiveness in the translation, since we believe existential types characterize the correct notion of abstraction.

The main question then that this paper addresses is under which circumstances is the dot notation equivalent to the open notation?

First, in section 2, we introduce a simple calculus with the open notation, intended to serve as a reference point. We give typechecking rules, denotational semantics, and show the soundness of the type system. In section 3, the `open` elimination construct is replaced by a simplified dot notation, consisting of two projections, `.Fst` and `.snd`, that are syntactically restricted to apply to simple variables only. Then we prove that the calculus thus obtained is equivalent to the former one by translating one calculus into the other. The translations are reasonably faithful, since they preserve both typing and semantics. Finally, in section 4, we lift the restriction on `.Fst` and `.snd`, and allow them to operate on any term; it seems that not all terms of this calculus with generalized dot notation have equivalents in the original calculus with `open`; however, we characterize a large class of terms that can be encoded in the open notation, while retaining typing and semantics.

2 A simple calculus with abstract types

The calculus in this section is based upon the usual simply typed λ-calculus, extended with constants and primitive operations as needed (such as integers, booleans and pairs, though we shall axiomatize only integers for simplicity.) To model type abstraction, we add existential quantification on types, and two term constructors: second-order dependent pairs and the `open` construct. These term constructors correspond to the introduction and elimination rules for existential quantification in constructive logic.

2.1 Syntax

In the following, X and Y range over a given countable set of type variable identifiers, A and B range over the class of types, similarly, x and y range over the set of term variable identifiers, and a, b, c range over the class of terms.

$$\begin{array}{rcl} A & ::= & X \mid A \to B \mid \exists X.\, A \\ a & ::= & x \mid \lambda x\!:\!A.\, b \mid b(a) \mid \langle X = A,\, b\!:\!B \rangle \mid \texttt{open}\ a\ \texttt{as}\ \langle X,\, y\!:\!B \rangle\ \texttt{in}\ b \end{array}$$

The construct $\langle X = A,\, b\!:\!B \rangle$ builds a second-order dependent pair. This is basically a pair of a type A and a term b of type B; however, b and B may depend on the binding $X = A$; that is, X may appear in b or in B, and is considered to be bound to A there. This binding is not visible outside of the pair; in particular, the type of the pair is simply $\exists X.\, B$, without any mention of A.

The elimination construct $\texttt{open}\ a\ \texttt{as}\ \langle X,\, y : B\rangle\ \texttt{in}\ b$ evaluates a to a pair, binds X to its internal type and y to its value, and returns the value of b in this new environment. The variable X gets bound in B and b; y is bound in b.

For some purposes, we also consider the calculus above extended with integer constants:

$$\begin{array}{rcl} A & ::= & \ldots \mid \texttt{Int} \\ a & ::= & \ldots \mid \texttt{0} \mid \texttt{succ}(a) \mid \texttt{case}\ a\ \texttt{of}\ \texttt{0}: b,\ \texttt{succ}(x): c \end{array}$$

The pattern-matching construct $\texttt{case}\ a\ \texttt{of}\ \texttt{0}: b,\ \texttt{succ}(x): c$ subsumes test for zero, the predecessor function, and the usual $\texttt{if}\ \ldots\ \texttt{then}\ \ldots\ \texttt{else}$ construct; x is considered bound (to the predecessor of a) in c.

As usual, expressions are identified up to a renaming of bound variables.

2.2 Typechecking

We specify the typing of terms of this calculus by a system of inference rules. The rules define the judgment "term a has type A under the assumptions E", written $E \vdash_o a : A$.

The assumptions are either "variable x has type A" or "identifier X is a valid type variable", hence the syntax of typing environments E is as follows:

$$E ::= \emptyset \mid E, X \mid E, x : A.$$

We write $Dom(E)$ for the set of variables introduced in E, that is:

$$Dom(\emptyset) = \emptyset \quad Dom(E, X) = Dom(E) \cup \{X\} \quad Dom(E, x : A) = Dom(E) \cup \{x\}$$

However, we must put additional constraints on the environments, to ensure that type variables are introduced in the environment before being used in types. So we have to define two more judgments: "typing environment E is well-formed", written $\vdash_o E$ ENV; and "type A is valid in environment E", written $E \vdash_o A$ TYPE, as follows.

$$\vdash_o \emptyset\ \textsc{env}$$

$$\frac{\vdash_o E\ \textsc{env} \qquad X \notin Dom(E)}{\vdash_o E, X\ \textsc{env}} \qquad \frac{E \vdash_o A\ \textsc{type} \qquad x \notin Dom(E)}{\vdash_o E, x : A\ \textsc{env}}$$

$$\frac{\vdash_o E\ \textsc{env}}{E \vdash_o \texttt{Int}\ \textsc{type}} \qquad \frac{\vdash_o E, X, E'\ \textsc{env}}{E, X, E' \vdash_o X\ \textsc{type}}$$

$$\frac{E \vdash_o A \text{ TYPE} \quad E \vdash_o B \text{ TYPE}}{E \vdash_o A \to B \text{ TYPE}} \qquad \frac{E, X \vdash_o A \text{ TYPE}}{E \vdash_o \exists X.\, A \text{ TYPE}}$$

We can now give the typechecking rules for terms. The first rules are exactly those of the simply typed λ-calculus:

$$\frac{\vdash_o E, x : A, E' \text{ ENV}}{E, x : A, E' \vdash_o x : A}$$

$$\frac{E \vdash_o A \text{ TYPE} \quad E, x : A \vdash_o b : B}{E \vdash_o \lambda x : A.\, b : A \to B}$$

$$\frac{E \vdash_o b : A \to B \quad E \vdash_o a : A}{E \vdash_o b(a) : B}$$

Typing rules for integers are straightforward:

$$\frac{\vdash_o E \text{ ENV}}{E \vdash_o \texttt{0} : \texttt{Int}} \qquad \frac{E \vdash_o a : \texttt{Int}}{E \vdash_o \texttt{succ}(a) : \texttt{Int}}$$

$$\frac{E \vdash_o a : \texttt{Int} \quad E \vdash_o b : A \quad E, x : \texttt{Int} \vdash_o c : A}{E \vdash_o \texttt{case}\ a\ \texttt{of}\ \texttt{0}: b,\ \texttt{succ}(x): c : A}$$

A pair $\langle X = A,\, b : B \rangle$ has type $\exists X.\, B$ if the claim $b : B$ holds when considering the binding $X = A$ in b and B. The type A does not appear in the type of the pair; it is therefore hidden outside of the pair.

$$\frac{E \vdash_o A \text{ TYPE} \quad E \vdash_o b\{X \leftarrow A\} : B\{X \leftarrow A\}}{E \vdash_o \langle X = A,\, b : B \rangle :\ \exists X.\, B}$$

If a has an existential type $\exists X.\, B$, then an $\texttt{open}\ a\ \texttt{as}\ \langle X,\, y : B \rangle\ \texttt{in}\ c$ construction has the same type as c. The term c is considered in an environment where the type variable X is defined and the term variable y has type B. Inside c, the type X is a formal variable, which cannot match any type except itself. To ensure that X does not escape the scope of the `open` expression, we require that X is not free in the type C of the body c. This is ensured by requiring that C is a valid type in the original environment E, in which X is unbound.

$$\frac{E \vdash_o a : \exists X.\, B \quad E \vdash_o C \text{ TYPE} \quad E, X, y : B \vdash_o c : C}{E \vdash_o \texttt{open}\ a\ \texttt{as}\ \langle X,\, y : B \rangle\ \texttt{in}\ c : C}$$

It is easy to see that $E \vdash_o a : A$ implies $E \vdash_o A$ TYPE, which implies $\vdash_o E$ ENV in turn. This fact is used implicitly in the rules above.

2.3 Evaluation

The usual approach to evaluation would be to define reduction rules on the typed terms; following Mitchell and Plotkin [13], we could take, for instance:

$$\begin{array}{rcl} (\lambda x\!:\!A.\, b)(a) & \to & b\{x \leftarrow a\} \\ \texttt{open}\ \langle X = A,\, b\!:\!B\rangle\ \texttt{as}\ \langle X,\, y\!:\!B\rangle\ \texttt{in}\ c & \to & c\{y \leftarrow b\}\{X \leftarrow A\} \end{array}$$

plus some rules for constants. However, these rules do not identify terms that intuitively have exactly the same meaning, for instance

$$\texttt{open}\ x\ \texttt{as}\ \langle Y,\, z\!:\!\texttt{Int}\rangle\ \texttt{in}\ \texttt{succ}(z)$$

and

$$\texttt{succ}(\texttt{open}\ x\ \texttt{as}\ \langle Y,\, z\!:\!\texttt{Int}\rangle\ \texttt{in}\ z).$$

Though the `open` a `as` ... construct performs very little computation, it is not allowed to disappear until a is reduced to an explicit pair, which may never occur. This prevents a number of desirable reductions: $(\texttt{open}\ x\ \texttt{as}\ \langle Y,\, z\!:\!Y\rangle\ \texttt{in}\ \lambda u\!:\!\texttt{Int}.\, u)(\texttt{0})$ is in normal form, while the corresponding untyped term $(\lambda z.\, \lambda u.\, u)(x)(\texttt{0})$ may be reduced. This turns out to be a major problem when trying to relate other calculi with this one, as we shall do later on, since the various typed reductions do not match, while the untyped reductions are the same.

We could add other reduction rules to perform the desired identifications, but it is unclear whether the Church-Rosser and strong normalization properties would still hold.

Instead, we choose to reduce the underlying untyped λ-terms, obtained by erasing all type annotations. Pairs of a type and a term are identified with the term itself, hence the pairing operation simply disappears, and `open` becomes a simple binding of a term to a variable, similar to the `let` construct of ML. For simplicity, we express it by mere substitution of the variable by the term.

$$\begin{array}{rcl} \mathit{Erase}_o(x) & = & x \\ \mathit{Erase}_o(\lambda x\!:\!A.\, b) & = & \lambda x.\, \mathit{Erase}_o(b) \\ \mathit{Erase}_o(b(a)) & = & \mathit{Erase}_o(b)(\mathit{Erase}_o(a)) \\ \mathit{Erase}_o(\langle X = A,\, b\!:\!B\rangle) & = & \mathit{Erase}_o(b) \\ \mathit{Erase}_o(\texttt{open}\ a\ \texttt{as}\ \langle X,\, y\!:\!A\rangle\ \texttt{in}\ b) & = & \mathit{Erase}_o(b)\{y \leftarrow \mathit{Erase}_o(a)\} \\ \mathit{Erase}_o(\texttt{0}) & = & \texttt{0} \end{array}$$

$$
\begin{array}{rcl}
\mathit{Erase}_o(\mathtt{succ}(a)) & = & \mathtt{succ}(\mathit{Erase}_o(a)) \\
\mathit{Erase}_o(\mathtt{case}\ a\ \mathtt{of}\ \mathtt{0}\colon b,\ \mathtt{succ}(x)\colon c) & = & \mathtt{case}\ \mathit{Erase}_o(a)\ \mathtt{of}\ \mathtt{0}\colon \mathit{Erase}_o(b), \\
 & & \qquad\qquad \mathtt{succ}(x)\colon \mathit{Erase}_o(c)
\end{array}
$$

After the erasing is performed, the untyped λ-term thus obtained is reduced in the usual way [1]. However, when we add constants to this calculus, there is a possibility of a run-time type error during reduction; for instance, an integer could be applied as if it were a function. It remains to show that this cannot occur if the initial term is well-typed.

To be more precise, we shall use a denotational semantics for the untyped λ-calculus. Following MacQueen, Plotkin and Sethi [10], we choose a domain $\mathbf{V}$ isomorphic to $\mathbf{N}_\perp + (\mathbf{V} \to \mathbf{V}) + \{\mathtt{wrong}\}_\perp$. Then to each untyped term m, considered in an environment ρ, we associate a meaning that is a value $[\![m]\!]_\rho$ of the "universe" $\mathbf{V}$; meaningless terms, that is terms whose evaluation leads to a type error, are mapped to `wrong`. The environment ρ is a partial mapping from term variables to values of $\mathbf{V}$.

$$
\begin{array}{rcl}
[\![x]\!]_\rho & = & \rho(x) \text{ if defined, } \mathtt{wrong} \text{ otherwise} \\
[\![\lambda x.m]\!]_\rho & = & v \mapsto [\![m]\!]_{\rho[x \leftarrow v]} \\
[\![m(n)]\!]_\rho & = & \text{if } [\![m]\!]_\rho \text{ is in } \mathbf{V} \to \mathbf{V} \text{ then } ([\![m]\!]_\rho)([\![n]\!]_\rho) \text{ else } \mathtt{wrong} \\
[\![\mathtt{0}]\!]_\rho & = & 0 \\
[\![\mathtt{succ}(m)]\!]_\rho & = & \text{if } [\![m]\!]_\rho \in \mathbf{N}_\perp \text{ then } [\![m]\!]_\rho + 1 \text{ else } \mathtt{wrong} \\
[\![\mathtt{case}\ m\ \mathtt{of}\ \mathtt{0}\colon n,\ \mathtt{succ}(x)\colon p]\!]_\rho & = & \text{if } [\![m]\!]_\rho = 0 \text{ then } [\![n]\!]_\rho \\
 & & \text{if } [\![m]\!]_\rho = i + 1 \text{ then } [\![p]\!]_{\rho[x \leftarrow i]} \\
 & & \text{else } \mathtt{wrong}
\end{array}
$$

We can now state the soundness of the typing rules:

Proposition 1 *For all terms a and types A, if $\emptyset \vdash_\circ a : A$, then $\mathit{Erase}_o(a)$ does not denote* `wrong`*, that is $[\![\mathit{Erase}_o(a)]\!]_\emptyset \neq \mathtt{wrong}$.*

To prove this proposition, we shall first give a meaning to type expressions as well. Following the ideal model of types [10], we interpret type expressions as ideals of $\mathbf{V}$, as follows:

$$
\begin{array}{rcl}
[\![\mathtt{Int}]\!]_\rho & = & \mathbf{N}_\perp \\
[\![X]\!]_\rho & = & \rho(X) \text{ if defined, } \emptyset \text{ otherwise} \\
[\![A \to B]\!]_\rho & = & \{f \in \mathbf{V} \to \mathbf{V} \mid \forall v \in [\![A]\!]_\rho, f(v) \in [\![B]\!]_\rho\} \\
[\![\exists X.\, A]\!]_\rho & = & \bigsqcup_{\substack{W \text{ ideal} \\ \mathtt{wrong} \notin W}} [\![A]\!]_{\rho[X \leftarrow W]}
\end{array}
$$

where in addition ρ maps type variables to ideals of $\mathbf{V}$. Notice that if `wrong` is not in $\rho(X)$ for any X, then $\mathtt{wrong} \notin [\![A]\!]_\rho$ for all types A. The soundness of typing then follows easily from the following claim, which is proved in [10]:

Proposition 2 *Assume $E \vdash_\circ a : A$. Let ρ be a mapping compatible with E; that is, for all type variables $X \in Dom(E)$, the ideal $\rho(X)$ does not contain* `wrong`*, and for all term variables $x \in Dom(E)$, the denotation $\rho(x)$ belongs to $[\![E(x)]\!]_\rho$. Then $[\![Erase_o(a)]\!]_\rho \in [\![A]\!]_\rho$.*

Apart from confirming that the type rules are sensible, this soundness result is a clue that indeed data abstraction is ensured in this calculus. Data abstraction is usually characterized by representation independence properties. These properties formalize the intuition that two "equivalent" implementations of an abstract type are not distinguishable; that is, the observed behavior of a program using one or the other is the same. Soundness of typing is a (weak) representation independence property, where two terms are equivalent when they have the same type, and observed behavior is the absence of run-time type errors. Stronger representation independence properties hold for this calculus; for instance, Mitchell [11] shows (for a superset of the calculus in this section) that if two implementations of an abstraction are related by a logical relation, then one can be substituted for the other in any closed term without modifying its meaning.

3 A calculus with the dot notation

We now formalize a calculus based on the dot notation. This is a λ-calculus with second-order dependent products, differing from the one in the previous section only in the way of splitting existentials: instead of a single `open` construct, it provides two constructs: one for accessing the value part of the pair, written x.`snd`; and one to get a witness of the abstracted type, written x.`Fst`. As this is intended to model the "qualified identifiers" of Modula-2 [16] and the "abstract tuples" of Quest [2], we shall in this section restrict .`Fst` and .`snd` to operate on term variables only.

3.1 Syntax

We keep the same notational conventions: X and Y are type variables, A and B are types, x and y are term variables, and a and b are terms.

$$\begin{array}{rcl} A & ::= & X \mid A \to B \mid \exists X.\, A \mid x.\texttt{Fst} \\ a & ::= & x \mid \lambda x\!:\!A.\, b \mid b(a) \mid \langle X = A,\, b\!:\!B \rangle \mid x.\texttt{snd} \end{array}$$

As previously, we can extend this dot calculus with integer constants:

$$\begin{array}{rcl} A & ::= & \ldots \mid \texttt{Int} \\ a & ::= & \ldots \mid \texttt{0} \mid \texttt{succ}(a) \mid \texttt{case}\ a\ \texttt{of}\ \texttt{0}\!:\!b,\ \texttt{succ}(x)\!:\!c \end{array}$$

3.2 Typechecking

As in the previous section, we have to define the judgment $E \vdash_d a : A$, as well as two auxiliary judgments $\vdash_d E$ ENV and $E \vdash_d A$ TYPE.

Type environments have exactly the same structure, and are well-formed under the same conditions :

$$\vdash_d \emptyset \text{ ENV}$$

$$\frac{\vdash_d E \text{ ENV} \qquad X \notin Dom(E)}{\vdash_d E, X \text{ ENV}} \qquad\qquad \frac{E \vdash_d A \text{ TYPE} \qquad x \notin Dom(E)}{\vdash_d E, x{:}A \text{ ENV}}$$

For type validity, we have to add one rule for the new construction $x.\texttt{Fst}$, stating that it is a valid type whenever x is shown to have an existential type.

$$\frac{\vdash_d E, X, E' \text{ ENV}}{E, X, E' \vdash_d X \text{ TYPE}} \qquad\qquad \frac{E, X \vdash_d A \text{ TYPE}}{E \vdash_d \exists X.\, A \text{ TYPE}}$$

$$\frac{E \vdash_d A \text{ TYPE} \qquad E \vdash_d B \text{ TYPE}}{E \vdash_d A \to B \text{ TYPE}} \qquad\qquad \frac{E \vdash_d x : \exists X.\, A}{E \vdash_d x.\texttt{Fst} \text{ TYPE}}$$

For typechecking of terms, we drop the `open` rule and keep the other four rules. As term variables may now appear in a type expression, they must be prevented from escaping their scope. Hence, the rule for functions $\lambda x{:}A.\, b$ states that x is not free in the type B of the body b. As in section 2.2, this is ensured by requiring that B is a valid type in the original environment E, which does not define x.

$$\frac{\vdash_d E, x{:}A, E' \text{ ENV}}{E, x{:}A, E' \vdash_d x : A}$$

$$\frac{E \vdash_d A \text{ TYPE} \qquad E \vdash_d B \text{ TYPE} \qquad E, x{:}A \vdash_d b : B}{E \vdash_d \lambda x{:}A.\, b : A \to B}$$

$$\frac{E \vdash_d b : A \to B \qquad E \vdash_d a : A}{E \vdash_d b(a) : B}$$

$$\frac{E \vdash_d A \text{ TYPE} \qquad E \vdash_d b\{X \leftarrow A\} : B\{X \leftarrow A\}}{E \vdash_d \langle X = A,\, b{:}B \rangle : \exists X.\, B}$$

The new construct $x.\texttt{snd}$ is well-typed provided that x has an existential type $\exists X.\, A$; its type is A, where the abstracted type X is substituted by its witness $x.\texttt{Fst}$.

$$\frac{E \vdash_d x : \exists X.\, A}{E \vdash_d x.\texttt{snd} : A\{X \leftarrow x.\texttt{Fst}\}}$$

3.3 Evaluation

As in section 2.3, we do not reduce typed terms directly, but rather strip all type information first and then evaluate the untyped λ-term thus obtained. This stripping is defined on terms with dot in the same vein as in section 2.3, with the additional case:

$$\mathit{Erase}_d(x.\texttt{snd}) = x$$

Instead of directly investigating this calculus — proving the soundness of the type system, for instance — we shall first try to relate it to the open calculus. Indeed, we are going to provide translations from terms of one calculus into the other calculus. The translations are reasonably faithful in that they preserve typing and semantics. Most interesting properties of the open calculus can then be effortlessly shown to hold in the dot calculus as well.

3.4 Encoding the dot calculus in the open calculus

The idea behind both translations is that, in the body of an $\texttt{open}\ x\ \texttt{as}\ \langle Y,\, z : A\rangle\ \texttt{in}\ b$ expression, Y and z seem to have the same meaning as $x.\texttt{Fst}$ and $x.\texttt{snd}$, respectively. This remark suggests the following strategy to transform a program with dot into one with `open`: insert some $\texttt{open}\ x\ \texttt{as}\ \langle Y,\, z : A\rangle\ \texttt{in}\ \ldots$ for each variable x used in a $x.\texttt{Fst}$ or $x.\texttt{snd}$ construct, and use Y and z instead of $x.\texttt{Fst}$ and $x.\texttt{snd}$. In other words:

$$b[x.\texttt{Fst}, x.\texttt{snd}] \longmapsto \texttt{open}\ x\ \texttt{as}\ \langle Y,\, z : A\rangle\ \texttt{in}\ b[Y, z]$$

For instance, this would allow the following translation:

$$\begin{array}{l}\lambda x : \exists X.\, X \times X \to \texttt{Int}.\ (\texttt{second}(x.\texttt{snd}))(\texttt{first}(x.\texttt{snd})) \longmapsto \\ \quad \lambda x : \exists X.\, X \times X \to \texttt{Int}.\ (\texttt{open}\ x\ \texttt{as}\ \langle Y,\, z : Y \times Y \to \texttt{Int}\rangle\ \texttt{in}\ \texttt{second}(z)) \\ \qquad\qquad\qquad\qquad (\texttt{open}\ x\ \texttt{as}\ \langle Y,\, z : Y \times Y \to \texttt{Int}\rangle\ \texttt{in}\ \texttt{first}(z))\end{array}$$

and this is obviously wrong, since the scoping constraint of `open` is not respected. (Y appears free in the types of the results.) So, this scheme must be restricted to the cases where $x.\texttt{Fst}$ does not appear in the type of $b[x.\texttt{Fst}, x.\texttt{snd}]$. Presumably, this may be achieved by taking a subexpression b large enough to enclose all uses of $x.\texttt{Fst}$ and $x.\texttt{snd}$;

$ComplexWRT(X)$ abbreviates
$(\texttt{Real} \to \texttt{Real} \to X) \times (X \to \texttt{Real}) \times (X \to \texttt{Real}) \times (X \to X \to X)$

In the dot calculus:
$\lambda x{:}\exists X.\ ComplexWRT(X).$
$\texttt{second}(x.\texttt{snd})\Big((\lambda z{:}x.\texttt{Fst}.\ \texttt{fourth}(x.\texttt{snd})(z)(z))(\texttt{first}(x.\texttt{snd})(3)(2))\Big)$

In the open calculus:
$\lambda x{:}\exists X.\ ComplexWRT(X).$
$\texttt{open}\ x\ \texttt{as}\ \langle X,\ y{:}ComplexWRT(X)\rangle\ \texttt{in}$
$\quad\texttt{second}(y)\Big((\lambda z{:}X.\ \texttt{fourth}(y)(z)(z))(\texttt{first}(y)(3)(2))\Big)$

Figure 5: Translation from the dot calculus to the open calculus

indeed, the body of the λ-abstraction binding x will do, since the typing rule for λ prevents x (and, hence, $x.\texttt{Fst}$) from being free in the type of the body.

Therefore, the following translation scheme seems sensible:

$$\lambda x{:}\exists X.\ A.\ b \longmapsto \lambda x{:}\exists X.\ A.\ \texttt{open}\ x\ \texttt{as}\ \langle X,\ y{:}A\rangle\ \texttt{in}\ b\{x.\texttt{Fst}{\leftarrow}X\}\{x.\texttt{snd}{\leftarrow}y\}$$

For instance, if we express the complex arithmetic example (figures 3 and 4) in the open calculus and in the dot calculus (see figure 5), the translation outlined above actually transforms the first program into the second one. This scheme works for closed terms, but we shall soon describe a translation for open terms as well.

Now, we would like to show that a well-typed term translates to a well-typed term. To allow for easy induction on derivations, we need to translate not only terms, but also whole judgments. Hence, we first reformulate the translation in a more general (and more controlled) way.

3.4.1 Formal definition of the translation

Let a_0 be a closed, well-typed term of the dot calculus. To avoid name clashes, we rename the identifiers appearing in a_0 so that none gets bound more than once.

Let $\mathcal{P}$ be the set of term variables x such that $x.\texttt{Fst}$ or $x.\texttt{snd}$ appear in a_0. Given the typechecking rules for projections, each $x \in \mathcal{P}$ has an existential type; we write it as $\exists T_x.\ A_x$, after renaming if necessary[2].

[2]Due to the typing rule for pairs, this type may differ from the type declared for x by the abstraction binding it, as in $\langle X = \exists Y.\ \texttt{Int},\ \lambda x : X.\ x.\texttt{snd} : X \to \texttt{Int}\rangle$. The type $\exists T_x.\ A_x$ we associate with x is the "true" type of x; that is, the type given to it in the derivation of $a_0 : A_0$ (this derivation is unique, given the typing rules), or alternatively the type declared for x, where all bindings such as $X = \exists Y.\ \texttt{Int}$ above have been performed.

To each $x \in \mathcal{P}$, we associate a term variable v_x that does not appear in a_0.
For each subterm a of a_0, we define its translation $\lfloor a \rfloor$ as follows:

$$\begin{array}{rcl}
\lfloor x \rfloor & = & x \\
\lfloor x.\texttt{snd} \rfloor & = & v_x \\
\lfloor \lambda x\!:\!A.\, b \rfloor & = & \lambda x\!:\!\lfloor A \rfloor.\, \texttt{open}\ x\ \texttt{as}\ \langle T_x,\, v_x\!:\!\lfloor A_x \rfloor \rangle\ \texttt{in}\ \lfloor b \rfloor \text{ if } x \in \mathcal{P} \\
\lfloor \lambda x\!:\!A.\, b \rfloor & = & \lambda x\!:\!\lfloor A \rfloor.\, \lfloor b \rfloor \text{ if } x \notin \mathcal{P} \\
\lfloor b(a) \rfloor & = & \lfloor b \rfloor(\lfloor a \rfloor) \\
\lfloor \langle X = A,\, b\!:\!B \rangle \rfloor & = & \langle X = \lfloor A \rfloor,\, \lfloor b \rfloor\!:\!\lfloor B \rfloor \rangle
\end{array}$$

This is basically the transformation outlined above, with substitutions delayed, and performed only when necessary: we insert an `open` x `as`... only when $x \in \mathcal{P}$; that is, when x.`Fst` or x.`snd` are actually used.

In the same vein, types translate as follows:

$$\begin{array}{rcl}
\lfloor X \rfloor & = & X \\
\lfloor x.\texttt{Fst} \rfloor & = & T_x \\
\lfloor A \to B \rfloor & = & \lfloor A \rfloor \to \lfloor B \rfloor \\
\lfloor \exists X.\, A \rfloor & = & \exists X.\, \lfloor A \rfloor
\end{array}$$

To translate environments, the only case that needs special treatment is the introduction of a variable x belonging to $\mathcal{P}$. In the derivation, this case corresponds to the typechecking of the body b of a function $\lambda x\!:\!A.\, b$. After translation, b will be preceded by an `open` x `as` $\langle T_x,\, v_x\!:\!\lfloor A_x \rfloor \rangle$ `in` ..., so we must typecheck the translation of b in an environment where T_x and v_x are defined. Hence the translation of $E, x\!:\!A$ introduces T_x and $v_x\!:\!\lfloor A_x \rfloor$ in addition to $x\!:\!\lfloor A \rfloor$.

$$\begin{array}{rcl}
\lfloor \emptyset \rfloor & = & \emptyset \\
\lfloor E, X \rfloor & = & \lfloor E \rfloor, X \\
\lfloor E, x\!:\!A \rfloor & = & \lfloor E \rfloor, x\!:\!\lfloor A \rfloor, T_x, v_x\!:\!\lfloor A_x \rfloor \text{ if } x \in \mathcal{P} \\
\lfloor E, x\!:\!A \rfloor & = & \lfloor E \rfloor, x\!:\!\lfloor A \rfloor \text{ if } x \notin \mathcal{P}
\end{array}$$

3.4.2 Preservation of typing

We are now able to translate any judgment componentwise, and in particular $\emptyset \vdash_d a_0 : A_0$. To prove that the translation preserves typing, it remains to show that its translation $\emptyset \vdash_o \lfloor a_0 \rfloor : \lfloor A_0 \rfloor$ can be derived in the open calculus.

Proposition 3 *Let a_0 be a closed, well-typed term of the dot calculus. Let $\lfloor\ \rfloor$ be the associated translation function. Let D be the derivation of $\emptyset \vdash_o a_0 : A_0$.*

- *If* $\vdash_d E$ ENV *is proved in* D*, then* $\vdash_o \lfloor E \rfloor$ ENV.
- *If* $E \vdash_d A$ TYPE *is proved in* D*, then* $\lfloor E \rfloor \vdash_o \lfloor A \rfloor$ TYPE.
- *If* $E \vdash_d a : A$ *is proved in* D*, then* $\lfloor E \rfloor \vdash_o \lfloor a \rfloor : \lfloor A \rfloor$.

Proof: By induction on the three derivations, starting with the one of $E \vdash_d a : A$.

- For value variables: we have the following derivation

$$\dfrac{\begin{array}{c}\vdots\\ \vdash_d E_1, x{:}A, E_2 \text{ ENV}\end{array}}{E_1, x{:}A, E_2 \vdash_d x : A}$$

  By induction, we have a derivation of $\vdash_o \lfloor E_1, x : A, E_2 \rfloor$ ENV. By definition of the translation, $\lfloor E_1, x : A, E_2 \rfloor$ has clearly the form $E_1', x : \lfloor A \rfloor, E_2'$. Hence the desired result $\lfloor E_1, x{:}A, E_2 \rfloor \vdash_o x : \lfloor A \rfloor$ follows.

- For value access in a pair: from the original derivation

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \vdash_d E_1, x{:}\exists T_x.\, A_x, E_2 \text{ ENV}\end{array}}{E_1, x{:}\exists T_x.\, A_x, E_2 \vdash_d x : \exists T_x.\, A_x}}{E_1, x{:}\exists T_x.\, A_x, E_2 \vdash_d x.\mathtt{snd} : A_x\{T_x{\leftarrow}x.\mathtt{Fst}\}}$$

  we get by induction a derivation of $\vdash_o \lfloor E_1, x : \exists T_x.\, A_x, E_2 \rfloor$ ENV. As $x.\mathtt{snd}$ appears in the derivation above, x belongs to $\mathcal{P}$. Therefore, $\lfloor E_1, x{:}\exists T_x.\, A_x, E_2 \rfloor$ introduces the variable v_x with type $\lfloor A_x \rfloor$. Hence $\lfloor E \rfloor \vdash_o v_x : \lfloor A_x \rfloor$, which implies the expected result

$$\lfloor E \rfloor \vdash_o \lfloor x.\mathtt{snd} \rfloor : \lfloor A\{X{\leftarrow}x.\mathtt{Fst}\} \rfloor$$

  since, as $\lfloor x.\mathtt{Fst} \rfloor = T_x$,

$$\lfloor A\{T_x{\leftarrow}x.\mathtt{Fst}\} \rfloor = \lfloor A \rfloor$$

- For abstraction:

$$\dfrac{\begin{array}{ccc}\vdots & \vdots & \vdots\\ E \vdash_d A \text{ TYPE} & E \vdash_d B \text{ TYPE} & E, x{:}A \vdash_d b : B\end{array}}{E \vdash_d \lambda x{:}A.\, b : A \to B}$$

If $x \notin \mathcal{P}$, by induction we have proofs of

$$\lfloor E \rfloor \vdash_o \lfloor A \rfloor \ \text{TYPE}$$

$$\lfloor E, x{:}A \rfloor \vdash_o \lfloor b \rfloor : \lfloor B \rfloor$$

that is

$$\lfloor E \rfloor, x{:}\lfloor A \rfloor \vdash_o \lfloor b \rfloor : \lfloor B \rfloor$$

hence

$$\lfloor E \rfloor \vdash_o \lambda x{:}\lfloor A \rfloor.\ \lfloor b \rfloor : \lfloor A \rfloor \to \lfloor B \rfloor$$

which is the desired result.

If $x \in \mathcal{P}$, then $A = \exists T_x.\ A_x$, and the same steps lead to:

$$\lfloor E \rfloor \vdash_o \exists T_x.\ \lfloor A_x \rfloor \ \text{TYPE}$$

$$\lfloor E \rfloor \vdash_o \lfloor B \rfloor \ \text{TYPE}$$

$$\lfloor E \rfloor, x{:}\exists T_x.\ \lfloor A_x \rfloor, T_x, v_x{:}\lfloor A_x \rfloor \vdash_o \lfloor b \rfloor : \lfloor B \rfloor$$

so we have the following derivation:

$$\dfrac{\dfrac{\begin{array}{c}\lfloor E \rfloor, x{:}\exists T_x.\ \lfloor A_x \rfloor \vdash_o x : \exists T_x.\ \lfloor A_x \rfloor \\ \lfloor E \rfloor \vdash_o \lfloor B \rfloor \ \text{TYPE} \\ \lfloor E \rfloor, x{:}\exists T_x.\ \lfloor A_x \rfloor, T_x, v_x{:}\lfloor A_x \rfloor \vdash_o \lfloor b \rfloor : \lfloor B \rfloor\end{array}}{\lfloor E \rfloor, x{:}\exists T_x.\ \lfloor A_x \rfloor \vdash_o \texttt{open}\ x\ \texttt{as}\ \langle T_x,\ v_x{:}\lfloor A_x \rfloor \rangle\ \texttt{in}\ \lfloor b \rfloor : \lfloor B \rfloor} \quad \lfloor E \rfloor \vdash_o \exists T_x.\ \lfloor A_x \rfloor \ \text{TYPE}}{\lfloor E \rfloor \vdash_o \lambda x{:}\exists T_x.\ \lfloor A_x \rfloor.\ \texttt{open}\ x\ \texttt{as}\ \langle T_x,\ v_x{:}\lfloor A_x \rfloor \rangle\ \texttt{in}\ \lfloor b \rfloor : \exists T_x.\ \lfloor A_x \rfloor \to \lfloor B \rfloor}$$

that leads to the desired result:

$$\lfloor E \rfloor \vdash_o \lfloor \lambda x{:}A.\ b \rfloor : \lfloor A \to B \rfloor$$

- For application: obvious by induction hypothesis.
- For pair construction: from

$$\dfrac{E \vdash_d A \ \text{TYPE} \qquad E \vdash_d b\{X{\leftarrow}A\} : B\{X{\leftarrow}A\}}{E \vdash_d \langle X = A,\ b{:}B \rangle : \exists X.\ B}$$

we get by induction hypothesis proofs of

$$\lfloor E \rfloor \vdash_o \lfloor A \rfloor \ \text{TYPE} \qquad \lfloor E \rfloor \vdash_o \lfloor b\{X{\leftarrow}A\} \rfloor : \lfloor B\{X{\leftarrow}A\} \rfloor$$

To conclude that $\lfloor E \rfloor \vdash_\circ \langle X = \lfloor A \rfloor,\ \lfloor b \rfloor : \lfloor B \rfloor \rangle : \exists X.\ \lfloor B \rfloor$, it suffices to show that

$$\begin{array}{rcl} \lfloor b\{X \leftarrow A\} \rfloor & = & \lfloor b \rfloor \{X \leftarrow \lfloor A \rfloor\} \\ \lfloor B\{X \leftarrow A\} \rfloor & = & \lfloor B \rfloor \{X \leftarrow \lfloor A \rfloor\} \end{array}$$

The proof is by induction on b and B. We give the non-trivial case: $b = \lambda x : C.\ d$ with $x \in \mathcal{P}$.

$$\begin{array}{rcl} \lfloor b\{X \leftarrow A\} \rfloor & = & \lambda x : \lfloor C\{X \leftarrow A\} \rfloor.\ \texttt{open}\ x\ \texttt{as}\ \langle T_x,\ v_x : \lfloor A_x \rfloor \rangle\ \texttt{in}\ \lfloor d\{X \leftarrow A\} \rfloor \\ & = & \lambda x : \lfloor C \rfloor \{X \leftarrow \lfloor A \rfloor\}.\ \texttt{open}\ x\ \texttt{as}\ \langle T_x,\ v_x : \lfloor A_x \rfloor \rangle\ \texttt{in}\ \lfloor d \rfloor \{X \leftarrow \lfloor A \rfloor\} \end{array}$$

by induction hypothesis. The type variable X is not free in A_x (indeed, by definition of T_x and A_x, the type $\exists T_x.A_x$ is C where some type variables have been substituted, and especially X has been substituted by A). Therefore X is not free in $\lfloor A_x \rfloor$ either. Hence the desired result:

$$\begin{array}{rcl} \lfloor b\{X \leftarrow A\} \rfloor & = & (\lambda x : \lfloor C \rfloor.\ \texttt{open}\ x\ \texttt{as}\ \langle T_x,\ v_x : \lfloor A_x \rfloor \rangle\ \texttt{in}\ \lfloor d \rfloor)\{X \leftarrow \lfloor A \rfloor\} \\ & = & \lfloor b \rfloor \{X \leftarrow \lfloor A \rfloor\} \end{array}$$

We turn now to the $E \vdash_d A$ TYPE judgment. All cases are straightforward, except maybe the one for $x.\texttt{Fst}$; but then the original derivation is:

$$\dfrac{\dfrac{\dfrac{\vdots}{\vdash_d E_1, x : \exists T_x.\ A_x, E_2\ \text{ENV}}}{E_1, x : \exists T_x.\ A_x, E_2 \vdash_d x : \exists T_x.\ A_x}}{E_1, x : \exists T_x.\ A_x, E_2 \vdash_d x.\texttt{Fst}\ \text{TYPE}}$$

The translation of $E_1, x : \exists T_x.\ A_x, E_2$ introduces the type variable T_x, and it is a well-formed environment, so $\lfloor E_1, x : \exists T_x.\ A_x, E_2 \rfloor \vdash_\circ T_x$ TYPE holds, which is the desired result since $\lfloor x.\texttt{Fst} \rfloor = T_x$.

Similarly, for the $\vdash_d E$ ENV judgment, the only interesting case is the introduction of an x belonging to $\mathcal{P}$. By definition of T_x and A_x, the type given to x must be $\exists T_x.\ A_x$.

$$\dfrac{\dfrac{\dfrac{\vdots}{E, T_x \vdash_d A_x\ \text{TYPE}}}{E \vdash_d \exists T_x.\ A_x\ \text{TYPE}} \qquad x \notin Dom(E)}{\vdash_d E, x : \exists T_x.\ A_x\ \text{ENV}}$$

By induction, we have a proof of $\lfloor E \rfloor, T_x \vdash_o \lfloor A_x \rfloor$ TYPE, hence the following derivation:

$$\dfrac{\dfrac{\dfrac{\lfloor E \rfloor, T_x \vdash_o \lfloor A_x \rfloor \text{ TYPE}}{\lfloor E \rfloor \vdash_o \exists T_x.\, \lfloor A_x \rfloor \text{ TYPE}} \quad x \notin Dom(\lfloor E \rfloor)}{\vdash_o \lfloor E \rfloor, x{:}\exists T_x.\, \lfloor A_x \rfloor \text{ ENV}} \quad T_x \notin Dom(\lfloor E \rfloor, x{:}\exists T_x.\, \lfloor A_x \rfloor)}{\vdash_o \lfloor E \rfloor, x{:}\exists T_x.\, \lfloor A_x \rfloor, T_x \text{ ENV}}$$

We can substitute this for the proof of $\vdash_o \lfloor E \rfloor, T_x$ ENV in the derivation of $\lfloor E \rfloor, T_x \vdash_o \lfloor A_x \rfloor$ TYPE, thereby obtaining a proof of $\lfloor E \rfloor, x{:}\exists T_x.\, \lfloor A_x \rfloor, T_x \vdash_o \lfloor A_x \rfloor$ TYPE. Hence:

$$\dfrac{\lfloor E \rfloor, x{:}\exists T_x.\, \lfloor A_x \rfloor, T_x \vdash_o \lfloor A_x \rfloor \text{ TYPE} \quad v_x \notin Dom(\lfloor E \rfloor, x{:}\exists T_x.\, \lfloor A_x \rfloor, T_x)}{\vdash_o \lfloor E \rfloor, x{:}\exists T_x.\, \lfloor A_x \rfloor, T_x, v_x{:}\lfloor A_x \rfloor \text{ ENV}}$$

□

3.4.3 Preservation of semantics

As the translation respects the structure of functions and applications, a closed term of the dot calculus and its translation in the open calculus have exactly the same underlying untyped terms. More precisely, we have:

Proposition 4 *For all subterms a of a_0,*

$$Erase_o(\lfloor a \rfloor)\{v_x \leftarrow x \text{ for all } x \in \mathcal{P}\} = Erase_d(a).$$

Hence $Erase_o(\lfloor a_0 \rfloor) = Erase_d(a_0)$.

Proof: By induction on a. We give the main cases:

- if $a = x.\mathtt{snd}$, $Erase_d(a) = x$ and $Erase_o(\lfloor a \rfloor) = v_x$.
- if $a = \lambda x{:}A.\, b$ and $x \in \mathcal{P}$, then

$$\begin{array}{rcl} Erase_o(\lfloor a \rfloor) & = & Erase_o(\lambda x{:}\lfloor A \rfloor.\, \mathtt{open}\ x\ \mathtt{as}\ \langle T_x,\, v_x{:}\lfloor A_x \rfloor \rangle\ \mathtt{in}\ \lfloor b \rfloor) \\ & = & \lambda x.\, (Erase_o(\lfloor b \rfloor)\{v_x \leftarrow x\}) \end{array}$$

  hence, by induction hypothesis,

$$\begin{array}{l} Erase_o(\lfloor a \rfloor)\{v_y \leftarrow y \text{ for all } y \in \mathcal{P}\} = \\ \quad \lambda x.\, (Erase_o(\lfloor b \rfloor)\{v_x \leftarrow x\}\{v_y \leftarrow y \text{ for all } y \in \mathcal{P} \setminus \{x\}\}) = \\ \quad \lambda x.\, Erase_d(\lfloor b \rfloor) \end{array}$$

  which is the expected result.

The remaining cases are obvious. □

Using the last two propositions, we can show the soundness of the type system of the dot calculus, extended with integer constants as mentioned in section 3.1. (The translation function is extended to the integer constructs in the trivial way, by translating their components recursively; it is easy to see that propositions 3 and 4 still hold.)

Corollary 1 *If* $\emptyset \vdash_d a_0 : A_0$*, then* $Erase_d(a_0)$ *does not denote* `wrong`.

Proof: The translation $\lfloor a_0 \rfloor$ is a well-typed closed term of the open calculus. Hence, by proposition 1, $[\![Erase_o(\lfloor a_0 \rfloor)]\!]_\emptyset \neq \texttt{wrong}$, which is the desired result, since $Erase_o(\lfloor a_0 \rfloor) = Erase_d(a_0)$. □

3.5 Encoding the open calculus in the dot calculus

The reverse translation is much less informative, but would confirm the intuition that the dot calculus is no less powerful than the open calculus, so we shall sketch it quickly.

The basic idea is to replace every `open` a `as` $\langle X,\, y{:}A\rangle$ `in` b by

$$(\lambda z{:}\exists X.\, A.\, b\{X{\leftarrow}z.\texttt{Fst}, y{\leftarrow}z.\texttt{snd}\})(a)$$

for some unused term variable z. In contrast with the previous translation, any type of the open calculus is also a type of the dot calculus, and this also holds for well-formed environments. Hence we just have to provide a translation for terms, which is straightforward:

$$\begin{array}{rcl}
\lceil x \rceil & = & x \\
\lceil \lambda x{:}A.\, b \rceil & = & \lambda x{:}A.\, \lceil b \rceil \\
\lceil b(a) \rceil & = & \lceil b \rceil(\lceil a \rceil) \\
\lceil \langle X = A,\, b{:}B \rangle \rceil & = & \langle X = A,\, \lceil b \rceil : B \rangle \\
\lceil \texttt{open}\ a\ \texttt{as}\ \langle X,\, y{:}A\rangle\ \texttt{in}\ b \rceil & = & (\lambda z{:}(\exists X.\, A).\, \lceil b \rceil\{X{\leftarrow}z.\texttt{Fst}, y{\leftarrow}z.\texttt{snd}\})(\lceil a \rceil) \\
 & & \text{where } z \text{ is not free in } \texttt{open}\ a\ \texttt{as}\ \langle X,\, y{:}A\rangle\ \texttt{in}\ b
\end{array}$$

We must check that, in the last rule, the substitution of y by $z.\texttt{snd}$ can be performed, since in $x.\texttt{Fst}$ or $x.\texttt{snd}$, x cannot be substituted by any term but another variable without producing a syntactically incorrect term such as $a(b).\texttt{Fst}$. Happily, by definition of the translation, if $y.\texttt{Fst}$ or $y.\texttt{snd}$ appears in $\lceil b \rceil$, then y is bound by a λ, and hence will not have to be substituted.

As $\lfloor\ \rfloor$, the function $\lceil\ \rceil$ preserves typing and semantics:

Proposition 5 *If* $E \vdash_o a : A$, *then* $E \vdash_d \lceil a \rceil : A$.

Proof: by induction on the original derivation. The only interesting case is the translation of an `open` construct:

$$\frac{E \vdash_o a : \exists X.\, A \qquad E \vdash_o B \text{ TYPE} \qquad E, X, y{:}A \vdash_o b : B}{E \vdash_o \texttt{open}\ a\ \texttt{as}\ \langle X,\ y{:}A\rangle\ \texttt{in}\ b : B}$$

By induction, it follows that $E \vdash_d \lceil a \rceil : \exists X.\, A$ and $E, X, y{:}A \vdash_d \lceil b \rceil : B$. By induction on the proof of the latter, we get a derivation of

$$E, z{:}\exists X.\, A \vdash_d \lceil b \rceil \{X \leftarrow z.\texttt{Fst}, y \leftarrow z.\texttt{snd}\} : B\{X \leftarrow z.\texttt{Fst}\}$$

As X is not free in B, we have $B\{X \leftarrow z.\texttt{Fst}\} = B$. Finally, from the proof of $E \vdash_d \lceil a \rceil : \exists X.\, A$, we can prove that $E \vdash_d \exists X.\, A$ TYPE. Putting all together, we get:

$$\frac{\dfrac{\begin{array}{c} E \vdash_d \exists X.\, A \text{ TYPE} \\ E \vdash_d B \text{ TYPE} \\ E, z{:}\exists X.\, A \vdash_d \lceil b \rceil \{X \leftarrow z.\texttt{Fst}, y \leftarrow z.\texttt{snd}\} : B \end{array}}{E \vdash_d \lambda z{:}\exists X.\, A.\, \lceil b \rceil \{X \leftarrow z.\texttt{Fst}, y \leftarrow z.\texttt{snd}\} : \exists X.\, A \to B} \qquad E \vdash_d \lceil a \rceil : \exists X.\, A}{E \vdash_d (\lambda z{:}\exists X.\, A.\, \lceil b \rceil \{X \leftarrow z.\texttt{Fst}, y \leftarrow z.\texttt{snd}\})(\lceil a \rceil) : B}$$

This is the expected result. □

Proposition 6 *For any term* a *of the open calculus,* $Erase_d(\lceil a \rceil)$ *β-reduces to* $Erase_o(a)$.

Proof: similar to the proof of proposition 4. □

4 A more powerful calculus with dot

In this section, we simply lift the restriction that only a value variable may be the argument of a `.Fst` or `.snd` construction, and allow instead any term, provided it has an existential type.

Regarding the formalism, this is a very natural generalization of the previous dot calculus, reminiscent of the second-order general sums (also called strong sums) of type theory [7, 5].

From the point of view of programming languages, this extension, in its full generality, does not seem to model any real situation, especially since `.Fst` now embeds the whole

class of values into the class of types; this means that the dividing line between types and values begins to blur dangerously.

However, we may feel the need for a calculus less restrictive than the dot calculus of the previous section. For instance, to deal with nested modules, it seems natural to have not only one-level access in modules, such as `module.Type`, but also access through paths of arbitrary length, such as `module.submodule.data`. To formalize this, the argument of a `.Fst` or `.snd` must be allowed to be a path, where a path is a term variable followed by an arbitrary number of `.snd`.

We could study this first extension of the dot calculus in the same way as for the dot calculus, by finding a translation to the open calculus and proving that it is faithful. However, other similar extensions may come to mind, and the same work would have to be done for each. Therefore, it seems easier to study the most general extension of all, where any term can appear to the left of a `.Fst` or `.snd` construct.

4.1 Typing

The typing rules are exactly those of the simple dot calculus, with the obvious generalization for the projections:

$$\frac{E \vdash_s a : \exists X.\, A}{E \vdash_s a.\mathtt{Fst}\ \text{TYPE}} \qquad \frac{E \vdash_s a : \exists X.\, A}{E \vdash_s a.\mathtt{snd} : A\{X \leftarrow a.\mathtt{Fst}\}}$$

It may be tempting to identify certain type expressions of this calculus, for instance $\langle X = A,\, b : B\rangle.\mathtt{Fst}$ and A, and to use this notion of type equivalence for typechecking, instead of for strict equality. This amounts to adding the following rule:

$$\frac{E \vdash_s a : A \qquad A \leftrightarrow B}{E \vdash_s a : B}$$

At first, we choose to disallow the latter rule and require syntactic equality for two types to be compatible; we shall come back to this issue later on.

4.2 Evaluation

As usual, we strip all type information before reducing. The `.snd` operation becomes identity on λ-terms:

$$\mathit{Erase}_s(a.\mathtt{snd}) = \mathit{Erase}_s(a)$$

As for the simple calculus with dot, we shall not investigate this calculus directly, but try to relate it to the open calculus first.

4.3 Relation to the open calculus

When we try to encode this calculus into the open calculus, we find terms which apparently have no equivalents in the open calculus. For instance, assume $a : A$ and consider the following term:

$$\langle X = A,\ \langle Y = X,\ a : Y\rangle.\texttt{snd} : \langle Y = X,\ a : Y\rangle.\texttt{Fst}\rangle$$

To express it in the open calculus, we have to insert an $\texttt{open}\ \langle Y = X,\ a : Y\rangle\ \texttt{as}\ldots$ at some point, and since X is free in the pair, the only literal translation is:

$$\langle X = A,\ \texttt{open}\ \langle Y = X,\ a : Y\rangle\ \texttt{as}\ \langle Z,\ z : Z\rangle\ \texttt{in}\ z : Z\rangle$$

but then Z escapes the scope of the `open` construct. (See later for a similar term but with no trivial redexes.)

To try to find an equivalent term in the open calculus, the general strategy is the same as in section 3.4: replace subterms such as $b[a.\texttt{Fst}, a.\texttt{snd}]$ by $\texttt{open}\ a\ \texttt{as}\langle X,\ y : A\rangle\ \texttt{in}\ b[X, y]$, with the additional constraints that $a.\texttt{Fst}$ should not appear in the type of $b[a.\texttt{Fst}, a.\texttt{snd}]$, and that b must not bind any of the free variables of a. However, the previous example shows that sometimes both constraints cannot be satisfied, especially when type variables are free in a.

If a has no free type variables, the transformation might work: let b be the body of the smallest abstraction $\lambda x : A.\, b$ enclosing all uses of $a.\texttt{Fst}$ and $a.\texttt{snd}$. Either $a.\texttt{Fst}$ does not appear in the type B of b, in which case it is possible to insert an $\texttt{open}\ a\ \texttt{as}\ldots$ there, or $a.\texttt{Fst}$ is part of B. But in the latter case, $x \notin FV(B)$, hence x is not free in a. Therefore, we can "lift" a out of b, consider the next enclosing λ, and iterate.

We are now going to formalize this argument in the same way as in the previous section by providing translations for terms, types, and environments.

4.3.1 A translation function

Let a_0 be a closed, well-typed term, renamed so that each variable gets bound once at most, and let D be the derivation of $\emptyset \vdash_s a_0 : A_0$. We write $\mathcal{P}$ for the set of subterms a of a_0 such that $a.\texttt{Fst}$ or $a.\texttt{snd}$ appears in D.

From now on, we shall suppose that the following condition holds:

(C) For all $a \in \mathcal{P}$, there are no type variables free in a.

For instance, it was not the case in the previous example, where $\mathcal{P} = \{\langle Y = X,\ a : Y\rangle\}$.

For each $a \in \mathcal{P}$, the term a has no free type variables, therefore D derives a type for a[3]; this type is an existential type (given the typechecking rules for .`Fst` and .`snd`),

[3]Hypothesis (C) is crucial here, since in general, it is not true that any subterm of a_0 is given a type in D, because of the typing rule for pairs $\langle X = A,\ b : B\rangle$, which requires that $b\{X \leftarrow A\}$ has a type, but not necessarily b itself. For instance, $\lambda x : X.\ \texttt{succ}(x)$ has no type by itself, though $\langle X = \texttt{Int},\ \lambda x : X.\ \texttt{succ}(x) : X \to \texttt{Int}\rangle$ is well-typed.

and we write $\exists T_a.\, A_a$ for it (after renaming of variables if necessary). Furthermore, we associate with a a value variable v_a unused in a_0.

Now we have to decide, for all $a \in \mathcal{P}$, where to insert an $\texttt{open}\ a\ \texttt{as}\ldots$ Let $a \in \mathcal{P}$. We consider the smallest subexpression b of a_0 binding all free variables of a. Since no variables are bound twice, such a b exists, and it contains all occurences of $a.\texttt{Fst}$ and $a.\texttt{snd}$. If a is closed, it is a_0. Otherwise, as there are no type variables free in a, it is the body of a λ-abstraction $\lambda x : A.\, b$. We shall consider only the latter case, since we can assume without loss of generality that a_0 itself is a λ-abstraction. We say that the variable x bound by this abstraction is the *first free variable* of a.

For all term variables x, we write $\mathcal{F}(x)$ for the sequence of all $a \in \mathcal{P}$ such that x is the first free variable of a. The translation of terms consists mainly in inserting, before the body b of any function $\lambda x : A.\, b$, an $\texttt{open}\ \lfloor a \rfloor\ \texttt{as}\ \langle T_a,\, v_a : \lfloor A_a \rfloor \rangle\ \texttt{in}\ \ldots$ for each $a \in \mathcal{F}(x)$. However, we must take care of the order of insertion, to avoid using a T_a or v_a before it is defined. More precisely, if a and a' belong to $\mathcal{F}(x)$ and a is a subexpression of a', then $\texttt{open}\ a\ \texttt{as}\ldots$ must precede $\texttt{open}\ a'\ \texttt{as}\ldots$ Therefore, we enumerate the sequence $\mathcal{F}(x) = a_1, \ldots, a_n$ in topological order, that is if a_i is a subexpression of a_j, then $i \leq j$.

$$
\begin{array}{rcl}
\lfloor x \rfloor & = & x \\
\lfloor a.\texttt{snd} \rfloor & = & v_a \\
\lfloor \lambda x : A.\, b \rfloor & = & \lambda x : \lfloor A \rfloor.\, \texttt{open}\ \lfloor a_1 \rfloor\ \texttt{as}\ \langle T_{a_1},\, v_{a_1} : \lfloor A_{a_1} \rfloor \rangle\ \texttt{in}\ \ldots \\
 & & \qquad \texttt{open}\ \lfloor a_n \rfloor\ \texttt{as}\ \langle T_{a_n},\, v_{a_n} : \lfloor A_{a_n} \rfloor \rangle\ \texttt{in}\ \lfloor b \rfloor \\
 & & \quad \text{if } \mathcal{F}(x) = a_1, \ldots, a_n \\
\lfloor b(a) \rfloor & = & \lfloor b \rfloor(\lfloor a \rfloor) \\
\lfloor \langle X = A,\, b : B \rangle \rfloor & = & \langle X = \lfloor A \rfloor,\, \lfloor b \rfloor : \lfloor B \rfloor \rangle
\end{array}
$$

Notice that the third rule remains valid if $\mathcal{F}(x) = \emptyset$, and means $\lfloor \lambda x : A.\, b \rfloor = \lambda x : \lfloor A \rfloor.\, \lfloor b \rfloor$ in this case. The translation of types is the same as for the original dot calculus:

$$
\begin{array}{rcl}
\lfloor X \rfloor & = & X \\
\lfloor a.\texttt{Fst} \rfloor & = & T_a \\
\lfloor A \to B \rfloor & = & \lfloor A \rfloor \to \lfloor B \rfloor \\
\lfloor \exists X.\, A \rfloor & = & \exists X.\, \lfloor A \rfloor
\end{array}
$$

As for the original dot calculus, we must synchronize the translation of environments with the translation of terms. Adding $x : A$ to E means that we are about to typecheck the body b of $\lambda x : A.\, b$. After translation, b will be preceded by $\texttt{open}\ a\ \texttt{as}\ \langle T_a,\, v_a : \lfloor A_a \rfloor \rangle\ \texttt{in}\ \ldots$ for each $a \in \mathcal{F}(x)$. So, the translation of $E, x : A$ must define T_a and $v_a : \lfloor A_a \rfloor$ for each $a \in \mathcal{F}(x)$.

$$
\lfloor \emptyset \rfloor \quad = \quad \emptyset
$$

$$\begin{array}{rcl} \lfloor E, X \rfloor & = & \lfloor E \rfloor, X \\ \lfloor E, x\!:\!A \rfloor & = & \lfloor E \rfloor, x\!:\!\lfloor A \rfloor, T_{a_1}, v_{a_1}\!:\!\lfloor A_{a_1} \rfloor, \ldots, T_{a_n}, v_{a_n}\!:\!\lfloor A_{a_n} \rfloor \\ & & \text{if } \mathcal{F}(x) = a_1, \ldots, a_n \end{array}$$

Condition (C) ensures that all $a \in \mathcal{P}$ are translated, and that no T_a or v_a is free in $\lfloor a_0 \rfloor$, since the `open` a `as` $\langle T_a, v_a\!:\!\lfloor A_a \rfloor\rangle$ `in` ... encloses all uses of T_a and v_a.

4.3.2 Preservation of typing

Proposition 7 *If one of the judgments* $\vdash_s E$ ENV, $E \vdash_s A$ TYPE, *or* $E \vdash_s a : A$ *is a step of the derivation* D, *then we can prove* $\vdash_o \lfloor E \rfloor$ ENV, $\lfloor E \rfloor \vdash_o \lfloor A \rfloor$ TYPE, *or* $\lfloor E \rfloor \vdash_o \lfloor a \rfloor : \lfloor A \rfloor$ *respectively.*

Proof: the proof is a straightforward generalization of the proof given for proposition 3. We proceed by induction on the derivation of the judgment. Here are the interesting cases:

- For value access in a pair: the original derivation is

$$\frac{\begin{array}{c}\vdots \\ E \vdash_s a : \exists T_a.\, A_a\end{array}}{E \vdash_s a.\mathtt{snd} : A_a\{T_a \leftarrow a.\mathtt{Fst}\}}$$

  so $a \in \mathcal{P}$ and E binds all free variables of a, including its first free variable. Hence, $\lfloor E \rfloor$ introduces v_a with the type $\lfloor A_a \rfloor$. By induction, we get a proof of $\lfloor E \rfloor \vdash_o \lfloor a \rfloor : \exists T_a.\, \lfloor A_a \rfloor$, from which we can extract a proof of $\vdash_o \lfloor E \rfloor$ ENV. Hence, $\lfloor E \rfloor \vdash_o v_a : \lfloor A_a \rfloor$, which is the expected result, since $\lfloor a.\mathtt{snd} \rfloor = v_a$ and obviously,

$$\lfloor A_a\{T_a \leftarrow a.\mathtt{Fst}\} \rfloor = \lfloor A_a \rfloor$$

- For λ-abstraction:

$$\frac{\begin{array}{ccc}\vdots & \vdots & \vdots \\ E \vdash_s A \text{ TYPE} & E \vdash_s B \text{ TYPE} & E, x\!:\!A \vdash_s b : B\end{array}}{E \vdash_s \lambda x\!:\!A.\, b : A \to B}$$

  Let $a_1, \ldots, a_n$ be $\mathcal{F}(x)$. For all i, since a_i is a subterm of b and since there are no type variables free in a_i, the derivation of $E, x\!:\!A \vdash_s b : B$ contains a derivation of $E_{a_i} \vdash_s a_i : \exists T_{a_i}.\, A_{a_i}$ for some environment E_{a_i}. So, by induction, we get proofs of

$$\lfloor E \rfloor \vdash_o \lfloor A \rfloor \text{ TYPE}$$

$$\lfloor E \rfloor \vdash_o \lfloor B \rfloor \text{ TYPE}$$

$$\Sigma_n \vdash_o \lfloor b \rfloor : \lfloor B \rfloor$$

$$\lfloor E_{a_i} \rfloor \vdash_o \lfloor a_i \rfloor : \exists T_{a_i}.\, \lfloor A_{a_i} \rfloor$$

where $\Sigma_j = \lfloor E \rfloor, x : \lfloor A \rfloor, T_{a_1}, v_{a_1} : \lfloor A_{a_1} \rfloor, \ldots, T_{a_j}, v_{a_j} : \lfloor A_{a_j} \rfloor$ for all $0 \leq j \leq n$.

As a_i is a subexpression of b, it is easy to see that Σ_n and $\lfloor E_{a_i} \rfloor$ bind the free variables of a_i to the same types. Furthermore, as $\mathcal{F}(x)$ is enumerated in topological order, for all $j \geq i$, the terms $a_j.\texttt{Fst}$ and $a_j.\texttt{snd}$ do not appear in a_i, therefore T_{a_j} and v_{a_j} are not free in $\lfloor a_i \rfloor$. Hence, Σ_{i-1} and $\lfloor E_{a_i} \rfloor$ bind the free variables of a_i to the same types. So, from the derivation of

$$\lfloor E_{a_i} \rfloor \vdash_o \lfloor a_i \rfloor : \exists T_{a_i}.\, \lfloor A_{a_i} \rfloor$$

we can build a proof of

$$\Sigma_{i-1} \vdash_o \lfloor a_i \rfloor : \exists T_{a_i}.\, \lfloor A_{a_i} \rfloor$$

Since Σ_i is an extension of $\lfloor E \rfloor$, from the proof of $\lfloor E \rfloor \vdash_o \lfloor B \rfloor$ TYPE, we get a proof of $\Sigma_i \vdash_o \lfloor B \rfloor$ TYPE. Hence the desired result:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma_n \vdash_o \lfloor b \rfloor : \lfloor B \rfloor \quad \Sigma_{n-1} \vdash_o \lfloor B \rfloor \text{ TYPE} \quad \Sigma_{n-1} \vdash_o \lfloor a_n \rfloor : \exists T_{a_n}.\, \lfloor A_{a_n} \rfloor}{\Sigma_{n-1} \vdash_o \texttt{open } \lfloor a_n \rfloor \texttt{ as } \langle T_{a_n},\, v_{a_n} : \lfloor A_{a_n} \rfloor \rangle \texttt{ in } \lfloor b \rfloor : \lfloor B \rfloor} \quad \Sigma_{n-2} \vdash_o \lfloor B \rfloor \text{ TYPE} \quad \Sigma_{n-2} \vdash_o \lfloor a_{n-1} \rfloor : \exists T_{a_{n-1}}.\, \lfloor A_{a_{n-1}} \rfloor}{\vdots}}{\lfloor E \rfloor, x : \lfloor A \rfloor \vdash_o \texttt{open } \lfloor a_1 \rfloor \texttt{ as } \langle T_{a_1},\, v_{a_1} : \lfloor A_{a_1} \rfloor \rangle \texttt{ in } \ldots \lfloor b \rfloor : \lfloor B \rfloor} \quad \lfloor E \rfloor \vdash_o \lfloor A \rfloor \text{ TYPE}}{\lfloor E \rfloor \vdash_o \lfloor \lambda x : A.\, b \rfloor : \lfloor A \to B \rfloor}$$

- For the introduction of a value variable in an environment:

$$\dfrac{E \vdash_s A \text{ TYPE} \qquad x \notin Dom(A)}{\vdash_s E, x : A \text{ ENV}}$$

Let $a_1, \ldots, a_n$ be $\mathcal{F}(x)$. If $n = 0$, we get by induction a proof of $\lfloor E \rfloor \vdash_o \lfloor A \rfloor$ TYPE, hence the desired result $\vdash_o \lfloor E \rfloor, x : \lfloor A \rfloor$ ENV.

Otherwise, by the same reasoning as in the case of λ-abstraction, we have a proof of:

$$\lfloor E \rfloor, x : \lfloor A \rfloor, T_{a_1}, v_{a_1} : \lfloor A_{a_1} \rfloor, \ldots, T_{a_{n-1}}, v_{a_{n-1}} : \lfloor A_{a_{n-1}} \rfloor \vdash_o \lfloor a_n \rfloor : \exists T_{a_n}.\ \lfloor A_{a_n} \rfloor$$

from which we can extract a derivation of:

$$\lfloor E \rfloor, x : \lfloor A \rfloor, T_{a_1}, v_{a_1} : \lfloor A_{a_1} \rfloor, \ldots, T_{a_{n-1}}, v_{a_{n-1}} : \lfloor A_{a_{n-1}} \rfloor \vdash_o \exists T_{a_n}.\ \lfloor A_{a_n} \rfloor \ \text{TYPE}$$

whose penultimate step is:

$$\lfloor E \rfloor, x : \lfloor A \rfloor, T_{a_1}, v_{a_1} : \lfloor A_{a_1} \rfloor, \ldots, T_{a_{n-1}}, v_{a_{n-1}} : \lfloor A_{a_{n-1}} \rfloor, T_{a_n} \vdash_o \lfloor A_{a_n} \rfloor \ \text{TYPE}$$

Hence the desired result:

$$\vdash_o \lfloor E \rfloor, x : \lfloor A \rfloor, T_{a_1}, v_{a_1} : \lfloor A_{a_1} \rfloor, \ldots, T_{a_n}, v_{a_n} : \lfloor A_{a_n} \rfloor \ \text{ENV}$$

The remaining cases are easy. □

4.3.3 Preservation of semantics

As in the previous section, this translation does not modify the meaning of the program:

Proposition 8 *For all subterms a of a_0,*

$$\mathit{Erase}_o(\lfloor a \rfloor)\{v_b \leftarrow \mathit{Erase}_o(\lfloor b \rfloor) \text{ for all } b \in \mathcal{P}\} = \mathit{Erase}_s(a).$$

Hence $\mathit{Erase}_o(\lfloor a_0 \rfloor) = \mathit{Erase}_s(a_0)$.

Proof: Same proof as for proposition 4. □

As a consequence of propositions 7 and 8, no closed term for which condition (C) holds can evaluate to `wrong`. Indeed, this is true even if condition (C) does not hold, since the type system of the generalized dot notation is sound, as can be proved directly along the lines of section 2.3. However, it is not clear whether the generalized dot notation ensures as strong an abstraction as the original `open` construct. Informally, we may fear that it is not the case, since, for instance, implementations of abstractions can be fully visible in types:

$$\langle X = \texttt{Int},\ 0 : X \rangle.\texttt{snd} : \langle X = \texttt{Int},\ 0 : X \rangle.\texttt{Fst}$$

thus publicizing that $X = \texttt{Int}$.

4.4 Type equivalence modulo reduction

As mentioned previously, we may add the following rule:

$$\frac{E \vdash_s a : A \qquad A \leftrightarrow B}{E \vdash_s a : B} \tag{1}$$

where the equivalence relation $\leftrightarrow$ is the congruence generated by the axioms:

$$\begin{array}{rcl} \langle X = A,\, b : B\rangle.\texttt{Fst} & \leftrightarrow & A \\ \langle X = A,\, b : B\rangle.\texttt{snd} & \leftrightarrow & b\{X \leftarrow A\} \\ (\lambda x : A.\, b)(a) & \leftrightarrow & b\{x \leftarrow a\} \end{array}$$

Even if rule 1 uses only equivalence between types, we need to define also equivalence between terms, because $a.\texttt{Fst}$ is equivalent to $b.\texttt{Fst}$ if a is equivalent to b.

This rule leads to a weaker notion of abstraction, where $\texttt{succ}(\langle X = \texttt{Int},\, \texttt{0} : X\rangle.\texttt{snd})$ typechecks, for instance. MacQueen [9] argues that this additional flexibility is desirable for a programming language, in order to be able to express complex dependencies between modules. However, the DL language he proposes is stratified, and this is not by chance, since an unstratified system like ours equipped with rule 1 is inconsistent, as it is possible to encode a calculus with a type of all types in it [12].

Even if we add some stratification, allowing reduction during typechecking, as above, does not seem to modify drastically the previous results. Admittedly, the counterexample we gave to show that some terms have no equivalent in the open calculus,

$$\langle X = A,\, \langle Y = X,\, a : Y\rangle.\texttt{snd} : \langle Y = X,\, a : Y\rangle.\texttt{Fst}\rangle,$$

now fails, since it reduces to $\langle X = A,\, a : X\rangle$. But we can work out more complicated examples, without any redex, that exhibit the same pathology, for instance:

$$\begin{array}{l} \lambda f : (\exists X.\, A) \to (\exists Y.\, B). \\ \quad \langle Z = A,\, (f\langle X = Z,\, a : X\rangle).\texttt{snd} : B\{Y \leftarrow (f\langle X = Z,\, a : X\rangle).\texttt{Fst}\}\rangle \end{array}$$

where $a : A$, since the $\texttt{open}\ \langle X = Z,\, a : X\rangle\ \texttt{as}\ldots$ must take place inside the $\langle Z = A,\, \ldots : \ldots\rangle$, hence violating the scoping rule for `open`.

Moreover, the translation above can still be applied to terms in normal form satisfying condition (C), and, in that case, still leads to well-typed terms of the open calculus, since for terms in normal form, type equivalence is again syntactic equality.

5 Conclusion

We have described two notations for type abstraction, one coming from logic, the other from programming, and investigated their relationships. This work contributes to the

formal foundation of the notion of abstraction found in programming. It also suggests some interesting extensions, as we shall see now.

The grammar of a calculus with the dot notation may ensure that condition (C) always holds. Let p range over the class of terms allowed to appear before a .`Fst` or a .`snd` construct:

$$\begin{array}{rcl} a & ::= & \ldots \mid p.\texttt{snd} \mid \ldots \\ A & ::= & \ldots \mid p.\texttt{Fst} \mid \ldots \end{array}$$

If p cannot contain any type variable at all, then condition (C) will certainly hold. This is the case, of course, for the simple dot calculus of section 2 ($p ::= x$).

This is also the case for the extension to "paths" mentioned at the beginning of the previous section ($p ::= x \mid p.\texttt{snd}$). This notion of paths gives a convenient way to deal with nested abstractions, which is much more natural than, for instance, the nested modules of Modula-2 [16]. Such a multi-level stratification (as opposed to the usual "flat" structure of programs considered as a set of modules) seems necessary for very large software systems.

Condition (C) even suggests other extensions, for instance:

$$p, p' ::= x \mid p.\texttt{snd} \mid p(p')$$

This would be convenient for dealing with parameterized abstractions. For example, assuming we extend the system with first-order dependent types $\forall(x : A)B$, we could write:

$$\begin{array}{l} \texttt{type}\ \mathit{Disc}(\mathit{cpx} : \mathit{Complex}) = \\ \quad \exists X.\ \mathit{make} : \mathit{cpx}.\texttt{Fst} \to \texttt{Real} \to X,\ \ldots \end{array}$$

$$\begin{array}{l} \texttt{val}\ \mathit{disc} : \forall(\mathit{cpx} : \mathit{Complex})\mathit{Disc}(\mathit{cpx}) = \\ \quad \lambda \mathit{cpx} : \mathit{Complex}. \\ \quad\quad \langle\ X \quad = \quad \mathit{cpx}.\texttt{Fst} \times \texttt{Real} \\ \quad\quad\quad \mathit{make} \quad = \quad \lambda \mathit{orig} : \mathit{Complex}.\ \lambda \mathit{radius} : \texttt{Real}.\ (\mathit{orig}, \mathit{radius}) \\ \quad\quad\quad \ldots\ \rangle \end{array}$$

Here, $\mathit{disc}(\mathit{cpx})$ would be a package implementing abstract discs of given origin and radius, depending on a given implementation cpx of $\mathit{Complex}$. For example, $\mathit{disc}(\mathit{polar\_complex})$ and $\mathit{disc}(\mathit{cartesian\_complex})$ would provide different implementations of the parametric *disc* abstraction. Two instances of $\mathit{disc}(\mathit{polar\_complex})$ would be recognized to refer to the same abstraction, according to the translation based on condition (C), and could hence interact freely.

In conclusion, we can define more and more expressive calculi based on the dot notation. The most expressive ones, only sketched in this section, should be able to deal with complex dependencies between first-class parametric abstractions. Our calculi are closer

to actual programming languages than are calculi based on logical notation, but enjoy all the same interesting properties.

Acknowledgements

We are indebted to Martín Abadi, for having pointed out several flaws in earlier proofs of propositions 3, 5, and 7.

References

[1] H. P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North-Holland, 1981.

[2] Luca Cardelli. *Typeful Programming*. Research Report 45, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto CA 94301. To appear in *Proc. IFIP State of the Art Seminar on Formal Description of Programming Concepts*, Rio de Janeiro, April 1989.

[3] Luca Cardelli, Peter Wegner. "On Understanding Types, Data Abstraction, and Polymorphism." *Computing Surveys*, 17(4), December 1985.

[4] John V. Guttag, James J. Horning, Jeannette M. Wing. "The Larch Family of Specification Languages." *IEEE Software*, September 1985.

[5] W. A. Howard. "The formulæ-as-types notion of construction." In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 479–490, Academic Press, 1980.

[6] Barbara Liskov, John Guttag. *Abstraction and Specification in Program Development*. The MIT Press, 1986.

[7] P. Martin-Löf. "Constructive mathematics and computer programming." *6th Int. Congress for Logic, Methodology, and Philosophy of Science*, pp. 153–175, North-Holland, 1982.

[8] D. B. MacQueen. "Modules for Standard ML." *ACM Symp. on Lisp and Functional Programming*, 1984.

[9] David MacQueen. "Using Dependent Types to Express Modular Structure." *13th Ann. ACM Symp. on Principles of Programming Languages*, 1986.

[10] David MacQueen, Gordon Plotkin, Ravi Sethi. "An Ideal Model for Recursive Polymorphic Types." *Information and Control* 71, pp. 95–130, 1986.

[11] John C. Mitchell. "Representation independence and data abstraction (preliminary version)." *13th Ann. ACM Symp. on Principles of Programming Languages*, 1986.

[12] John C. Mitchell, Robert Harper. "The Essence of ML." *15th Ann. ACM Symp. on Principles of Programming Languages*, 1988.

[13] John C. Mitchell, Gordon D. Plotkin. "Abstract types have existential type." *11th Ann. ACM Symp. on Principles of Programming Languages*, 1984.

[14] D. L. Parnas. "On the criteria to be used in decomposing systems into modules." *Communications of the ACM*, Vol. 15, No. 12, pp. 1053–1058, December 1972.

[15] J. C. Reynolds. "Towards a theory of type structure." *Colloquium sur la programmation*, Lecture Notes in Computer Science Vol. 19, pp. 408-425, Springer-Verlag, 1974.

[16] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 1983.