Choice in Dynamic Linking

Martín Abadi[1], Georges Gonthier[2], and Benjamin Werner[3]

[1] University of California at Santa Cruz
[2] Microsoft Research
[3] INRIA – Futurs and LIX, Projet LogiCal, Pôle Commun de Recherche en Informatique du Plateau de Saclay, INRIA, CNRS, E. Polytechnique et U. Paris-Sud

Abstract. We introduce a computational interpretation for Hilbert's choice operator (ε). This interpretation yields a typed foundation for dynamic linking in software systems. The use of choice leads to interesting difficulties—some known from proof theory and others specific to the programming-language perspective that we develop. We therefore emphasize an important special case, restricting the nesting of choices. We define and investigate operational semantics. Interestingly, computation does not preserve types but it is type-sound.

1 Introduction

In the 1920s, Hilbert invented the choice operator ε as a means of defining the first-order universal and existential quantifiers in an attempt to establish the consistency of arithmetic and analysis. Usually, in first-order logical systems, if A is a formula and x is a variable then $\varepsilon x.A$ is a term that represents some element x for which A holds, when such an x exists. The term $\varepsilon x.A$ is syntactically well-formed even when no such x exists. Hence, $\exists x.A$ may be regarded as an abbreviation for $A[(\varepsilon x.A)/x]$. (See section 7 for some references on ε.)

In this paper, we introduce a computational interpretation of the choice operator ε in the context of a second-order propositional logic, that is, in a variant of propositional ε-calculus. Its originality stems from the view of this operator as a construct in the type system for a programming language. In our type system, ε binds a type variable—rather than a variable that ranges over values—much like $\forall$ in the polymorphic λ-calculus System F [9,3]. If A is a type and X is a type variable then $\varepsilon X.A$ is a type X for which A is inhabited, when such an X exists. The type X may be chosen dynamically (at run-time) among several candidates. In any case, X is unique. For instance, $\varepsilon X.X$ is an arbitrary, fixed inhabited type, and if $\bot$ is an empty type, then $\varepsilon X.\bot$ is an arbitrary, fixed type.

Our programming-language perspective has substantial consequences. In particular, it constrains orders of program evaluation. We cannot blindly rely on analogues of the strategies previously explored in the proof theory for the choice operator (e.g., [14]): these strategies are generally not attractive for the operational semantics of programs.

While some of the logical difficulties caused by the choice operator are fairly well-known, we find others in this context. In short, we observe that ε tends to

I. Walukiewicz (Ed.): FOSSACS 2004, LNCS 2987, pp. 12–26, 2004.
© Springer-Verlag Berlin Heidelberg 2004

conflict with type-soundness, parametricity, and termination, and (non-conservatively) extends typed functional programming with tricky side effects. Some of these issues are crucially affected by orders of program evaluation.

In light of these difficulties, we particularly focus on an important special case in which ε's cannot be nested arbitrarily. With such restrictions, we define and investigate operational semantics. Interestingly, computation does not preserve types but it is type-sound. For example, a step of computation might replace the type $\varepsilon X.X$ with the type Bool. Such instantiations result in global changes of types, but—if done with great care—not in run-time type errors.

Choice is obviously related to existential quantification, and thereby [19] to abstract datatypes. From a programming-language perspective, choice enables us to refer to uniform, unique implementations of abstract datatypes. Two occurrences of $\varepsilon X.A$ in a program always refer to the same type X. In contrast, two occurrences of $\exists X.A$ in a program will typically yield different, incompatible concrete representations for X; such incompatibility is a well-known source of problems, for example in the treatment of binary methods. Programming languages provide several other ways of overcoming or avoiding those problems (for instance, the "dot notation" [4]). In contrast with many programming-language inventions, choice remains fruitfully close to logic.

At the same time, choice seems intriguingly close to practice. Specifically, whenever we instantiate $\varepsilon X.A$ with a chosen type B, we also pick a value of the corresponding type $A[B/X]$. In programming terms, this value can be seen as a dynamically linked implementation of the interface A, with B as the concrete representation type for X. Further, consecutive instantiations of a type variable correspond to incremental implementations of an interface.

As a result of our exploration, we therefore obtain a foundation for (aspects of) typed dynamic linking in extensible software systems (e.g., [2,10,6,7,12]). Dynamic linking has thus far been rather mysterious and notoriously error-prone (e.g., [5]). It has often been defined rather vaguely, or kept "under the covers". We hope that studies such as ours will contribute to taming it.

The next section describes the syntax and type system of a minimal programming language with choice, which we call System $\mathcal{E}$. Section 3 starts an analysis of the possible computation rules for this language and of some of the difficulties involved. Section 4 focuses on an important fragment, System $\mathcal{E}^*$, that has a simple and sound operational semantics. Section 5 treats an example. Section 6 briefly considers parametricity, termination, and conservativity. Finally, section 7 concludes with a discussion of related and further work. Because of space constraints, this paper omits some further analysis of computation rules for System $\mathcal{E}$; it also omits material on abstract machines, which are the subject of ongoing work.

2 System $\mathcal{E}$: Basics

This section describes the syntax, type system, and informal semantics of System $\mathcal{E}$, postponing formal semantics.

2.1 Defining System $\mathcal{E}$

Design. For simplicity and in order to focus on the choice operator, our programming language is a rather spartan λ-calculus. Since our choice operator binds type variables, we may wonder how much higher-order machinery the programming language should include. In particular, we may ask whether the type quantifier $\forall$ should be primitive, as in System F [9,3]. The interaction of ε with $\forall$ seems to raise many interesting but non-trivial questions, as we hint in section 6.2. Moreover, the full power of System F quantification is rarely present in current practical languages. Therefore, for this first study of ε in a programming context, we omit $\forall$. We also omit higher-order type operators, recursion, subtyping, and mutable references. We even omit base types (Bool, ...) and first-order types ($A \times B$, ...), but we liberally rely on them when appropriate (for instance, in examples); they are not problematic.

Syntax. Thus, System $\mathcal{E}$ is an extension of the simply typed λ-calculus, without base types but with ε at the level of types and corresponding implementations at the level of terms. Its grammar is:

$A, B, T ::=$	types
$\quad X$	$\quad$ type variable
$\quad A \to B$	$\quad$ function type
$\quad \varepsilon X.A$	$\quad$ choice
$e, t, u ::=$	terms
$\quad x$	$\quad$ variable
$\quad \lambda x{:}A.t$	$\quad$ function
$\quad t\ u$	$\quad$ application
$\quad \langle t : A \text{ with } X = T\rangle$	$\quad$ implementation

The type variable X is bound in $\varepsilon X.A$ and in $\langle e : A \text{ with } X = T\rangle$, with A as scope. We do not detail the usual definition of substitution for terms and types, respectively written $t[u/x]$ and $A[T/X]$.

Informal semantics. The intended meaning of the constructs borrowed from the simply typed λ-calculus is standard. We adopt a call-by-value interpretation. As explained in the introduction, $\varepsilon X.A$ is a type X for which A is inhabited, when such an X exists. We think of A as an interface in which X stands for a representation type. The type $\varepsilon X.A$ may be chosen dynamically. When an expression $\langle t : A \text{ with } X = T\rangle$ is evaluated, it fixes $\varepsilon X.A$ to be T, accordingly fixes the code for A to be t (locally and elsewhere), and executes t. The details of this process are quite delicate, and may become clear only with formal semantics.

Abbreviations. We abbreviate $\langle e : A \text{ with } X = T\rangle$ to $\langle e : A\rangle$ when X does not occur free in A (and in that case T can be arbitrary). We write $\langle e : A \text{ with } T\rangle$ for $\langle e : A \text{ with } X = T\rangle$ when X is clear from context. Similarly, when X is clear from context, we write $A[T]$ for $A[T/X]$, and εA for $\varepsilon X.A$. We write $\mathsf{let}\ x = t\ \mathsf{in}\ u$ for $(\lambda x{:}T.u\ t)$, when T is clear from context. We further write $t; u$ for $\mathsf{let}\ x = t\ \mathsf{in}\ u$ when x does not occur free in u.

Typing with ε. A context Γ is a finite set of pairs $(x : T)$ of variables and types. Typing judgements are of the form $\Gamma \vdash t : T$ and can be derived by the usual rules of the simply typed λ-calculus extended with the following new rule for ε:

$$\frac{\Gamma \vdash e : A[T/X]}{\Gamma \vdash \langle e : A \text{ with } X = T\rangle : A[(\varepsilon X.A)/X]}$$

This typing rule corresponds to the regular inference rule for ε in first-order logic. In natural-deduction style, that rule is:

$$\frac{A[t/x]}{A[(\varepsilon x.A)/x]}$$

where A ranges over formulas and t over terms. This rule strongly resembles the typing rule, but omits the typing environment (which corresponds to assumptions higher in the natural-deduction proof tree) and the expressions (which embody the proofs). Such similarities are the norm whenever the Curry-Howard isomorphism connects a logic and a type system.

Alternative typing rules. Several alternative typing rules are worth mentioning. A minor variant of our new rule includes a type substitution on terms:

$$\frac{\Gamma \vdash e[T/X] : A[T/X]}{\Gamma \vdash \langle e : A \text{ with } X = T\rangle : A[(\varepsilon X.A)/X]}$$

This variant might be convenient, because it can result in more compact expressions, but is not essential. A more significant alternative consists in extending the syntax and the typing rule so that one can choose several types at once. We return to such simultaneous choices in section 4.3.

2.2 Examples

We close this section with a brief, informal look at a few examples. We consider further examples below, also discussing their operational semantics.

Suppose that several software components rely on auxiliary compression packages that provide string compression and decompression functions. Many of the components may come with their own compression packages, each with a different internal representation for compressed data. One may prefer for all the components to rely on the same package, so that they can exchange compressed data. Any one of the possible packages may do. There may not be a convenient way to predict that such a package will be needed and to pick one, a priori, but the first use of such a package could trigger the loading of one implementation. For this purpose, we define the type expressions:

$$\begin{aligned} \mathsf{CompressPkg} &= \{c : \mathsf{String} \to X, d : X \to \mathsf{String}\} \\ \mathsf{Compressed} &= \varepsilon X.\mathsf{CompressPkg}[X] \end{aligned}$$

where we write $\{c : \mathsf{String} \to X, d : X \to \mathsf{String}\}$ for the type of records with c and d components with respective types $\mathsf{String} \to X$ and $X \to \mathsf{String}$. One component, e, may use a trivial compression package f:

$$\begin{aligned} f &= \{c = \lambda s{:}\mathsf{String}.s, d = \lambda s{:}\mathsf{String}.s\} \\ e &= \mathsf{let}\ h = \langle f : \mathsf{CompressPkg} \text{ with } X = \mathsf{String}\rangle\ \mathsf{in} \\ &\quad \ldots h.c \ldots h.d \ldots \end{aligned}$$

Another component, e', may use a more interesting compression package f' in which natural numbers implement compressed data:

$$\begin{aligned} f' &= \{c = \mathsf{string2nat}, d = \mathsf{nat2string}\} \\ e' &= \mathsf{let}\ h = \langle f' : \mathsf{CompressPkg} \text{ with } X = \mathsf{Nat}\rangle\ \mathsf{in} \\ &\quad \ldots h.c \ldots h.d \ldots \end{aligned}$$

Now e and e' may be combined in a larger expression m. For example, if e has type $\mathsf{Compressed}$ and e' has type $\mathsf{Compressed} \to \mathsf{Nat}$, we may write $m = (e'\ e)$ with type Nat. At run-time, the execution of m loads f or f' but does not mix them, avoiding type errors.

Similarly, consider the interface:

$$\mathsf{NatList} = \{\mathsf{nil} : X,\ \ \mathsf{cons} : \mathsf{Nat} \to X \to X,\ \ \mathsf{member} : X \to \mathsf{Nat} \to \mathsf{Bool}\}$$

for lists of natural numbers, with an empty list and with cons and membership operations. When t and u are two expressions of type $\mathsf{NatList}[\varepsilon X.\mathsf{NatList}]$, they may rely on different internal representations for lists, but it is still safe to write terms such as $t.\mathsf{cons}(2)(u.\mathsf{nil})$. At run-time, only one internal representation is used.

A deeper and more detailed example is given in section 5, including operational semantics.

3 Computing in System $\mathcal{E}$

Much as in the study of control operators, which correspond to classical logic, we aim to explore a new programming-language construct that corresponds to an extension of intuitionistic logic with ε. From a logical perspective, the aim is a cut-elimination result as general as possible. This approach was the one taken by Leisenring [14] in a first-order framework, with backtracking over choices and reduction under binders (strong reduction, in programming-language terms). In contrast, we favor a programming perspective, focusing on implementable and predictable operational semantics. However, our approach is also relevant to the more exotic strategies that arise from logical cut-elimination.

3.1 Linking

In order to perform computation with a System $\mathcal{E}$ term t, we must be able to replace an implementation term $\langle e' : A \text{ with } T'\rangle$ that appears in head position

in t with some term e. This step is essentially a linking operation: replacing a link to a potential implementation e' of A with an actual implementation e (which can be, but is not necessarily e').

More precisely, let t be a term, and $m = \langle e : A \text{ with } T\rangle$ an implementation of the interface $A[\varepsilon A]$. Suppose that some implementation m' of $A[\varepsilon A]$, $m' = \langle e' : A \text{ with } T'\rangle$, occurs in t. Linking m' to m, that is, replacing m' with e, implies linking the interface choice type εA to the type T used for εA in e, that is, replacing εA with T. (This replacement also applies to any types α-convertible to εA.)

However, there is no scope for εA: the substitution of T for εA must be global to t. In contrast, for existential types, the elimination construct for $\exists A$ (named *open* in [3]) delineates statically the scope in which the chosen representation type T can be used. There is no such elimination construct for $A[\varepsilon A]$. Thus, we may view $A[\varepsilon A]$ as the open interface type for the interface A, and $\exists A$ as the closed one.

Because the substitution of T for εA is global, it reaches all other implementations of $A[\varepsilon A]$ in t. In order to interoperate with e, those implementations must, at the very least, use the same representation type T as e. The only way of achieving this effect in practice is to link them with m as well. Thus, the entire linking operation must be global.

Definition 1. *Let again m stand for $\langle e : A \text{ with } T\rangle$. The* static linking *of t with m, noted $t \wr m$, is defined as the term obtained by simultaneously replacing all*[1] *implementations of $A[\varepsilon A]$ with e, and all instances of εA with T. If we use $\star$ as a wildcard in pattern-matching, we can write this as:*

$$t \wr \langle e : A \text{ with } T\rangle = t[T/\varepsilon A, e/\langle \star : A \text{ with } \star\rangle]$$

We extend the linking notation to types and typing contexts:

$$R \wr m = R[T/\varepsilon A] \qquad\qquad \Gamma \wr m = \Gamma[T/\varepsilon A]$$

One may argue that the linking operation, as specified here, is unsatisfactory for practical, efficient implementations of programming languages. In particular, it requires testing type equalities at run-time. Section 4.3 offers a remedy.

3.2 Type Soundness

In spite of the precautions we have taken, $t \wr m$ is not always well-typed. Let

$$C = X \to Y \qquad B = \varepsilon X.\varepsilon Y.C \qquad B' = \varepsilon Y.C[B/X]$$

and consider the term

$$u = \langle \mathsf{not} : \mathsf{Bool} \to Y \text{ with } Y = \mathsf{Bool}\rangle$$

[1] That is, of course, except the implementations of $A[\varepsilon A]$ occurring inside e. The example of section 5 illustrates this point.

which has type

$$\mathsf{Bool} \to \varepsilon Y.C[\mathsf{Bool}/X]$$

It follows that the type of

$$v = \langle (u\ \mathsf{true}) : \varepsilon Y.C \text{ with } X = \mathsf{Bool}\rangle$$

is B', but B' does not appear in v. Now

$$w = \langle \lambda x{:}B.1 : B \to Y \text{ with } Y = \mathsf{Nat}\rangle$$

is an implementation of type $C[B/X][B'/Y]$, so $((\lambda x{:}B'.x)\ v) \wr w = ((\lambda x{:}\mathsf{Nat}.x)\ v)$ is not well-typed.

This difficulty is well-known in the proof theory of ε. It arises when there are so-called "improper" ε type expressions that contain ε subexpressions in which the outer ε type variable appears free. For example, $\varepsilon X.\varepsilon Y.(X \to Y)$ is improper because the outer type variable X occurs under εY.

More precisely, we say that a type εC is *subordinate* in an implementation $\langle e : B \text{ with } Y = T\rangle$ if B contains a subexpression $\varepsilon B'$, in which Y appears (making $\varepsilon Y.B$ improper), such that εC is equal to either $\varepsilon B'[T/Y]$ or $\varepsilon B'[\varepsilon Y.B/Y]$; we say that εC is subordinate in t if εC is subordinate in some subterm of t.

By a straightforward structural induction, we obtain a conditional type-soundness result:

Theorem 1. *Suppose that $\Gamma, \Delta \vdash t : R$ and $\Gamma, \Delta' \vdash m : A[\varepsilon A]$, where Γ, Δ, and Δ' are disjoint contexts. If εA is not subordinate in t and does not appear in Γ, then*

$$\Gamma, \Delta \wr m, \Delta' \vdash t \wr m : R \wr m$$

The difficulties explained above make it impossible to remove the conditions of this statement in order to obtain an unconditional type-soundness theorem. One might expect that a solution could be based on work in proof theory, and in particular on Leisenring's approach. Unfortunately, that approach is not directly suitable in a programming context. We have found realistic examples with dependent choices in which "normal" orders of evaluation lead to type errors, and in which orders of evaluation based on Leisenring's are inappropriate. We may go further in two ways:

- One is to choose an evaluation order which avoids subordinate interface types. Such evaluation orders may be too complex for programming purposes. We omit their description.
- The other is to avoid subordinate interface types entirely. The corresponding restrictions seem reasonable, and we describe one next, in section 4.

4 System $\mathcal{E}^*$

In this section we start to explore a relatively simple but important fragment of System $\mathcal{E}$ that we call System $\mathcal{E}^*$. We believe that this fragment is useful in its own right (not only as a possible stepping stone), so a substantial part of this paper is devoted to its development.

4.1 Defining System $\mathcal{E}^*$

System $\mathcal{E}^*$ is a well-behaved fragment of System $\mathcal{E}$ that forbids interleaved ε binders in types. From a logical perspective, it corresponds to what Leisenring called the ε^*-calculus. From a programming-language perspective, this means that open interfaces cannot occur in signatures.

Definition 2. *A type is valid in System $\mathcal{E}^*$ (in short, is in $\mathcal{E}^*$) if it verifies the two properties:*

- *all its subexpressions are in $\mathcal{E}^*$,*
- *if it is of the form $\varepsilon X.A$, if $\varepsilon Y.B$ is a subexpression of A, then there is no occurrence of X in B which is free in A.*

A term t or context Γ is in $\mathcal{E}^*$ if every type occurring in it is in $\mathcal{E}^*$.

Note that we can obtain the same restriction by requiring that ε always binds the same fixed type variable (very roughly analogous to self or like Current for objects).

For example, the type $\varepsilon X.(X \to \varepsilon Y.(Y \to Y))$ is in $\mathcal{E}^*$ but $\varepsilon X.(X \to \varepsilon Y.(X \to Y))$ is not. The former type can be written with a single bound type variable, as $\varepsilon X.(X \to \varepsilon X.(X \to X))$. The latter type cannot be rewritten analogously, because of the occurrence of a bound variable underneath another binder.

For types in $\mathcal{E}^*$, the typing rules for System $\mathcal{E}^*$ are identical to those of System $\mathcal{E}$. Crucially, in System $\mathcal{E}^*$, no type is subordinate in any implementation. Therefore, Theorem 1 always applies. Furthermore, if Γ, t, and m are valid, then so are $\Gamma \wr m$ and $t \wr m$. Thus, the way is paved for simple operational semantics for System $\mathcal{E}^*$. This system remains rich enough for many examples, such as those developed in sections 2.2 and 5.

4.2 Operational Semantics

Basically, we choose a functional evaluation strategy (say, here, right-to-left call-by-value) and we perform a static linking step, on the fly, each time an implementation is encountered.

More formally, it is convenient to use evaluation contexts. We adopt the following, usual, definitions for values and evaluation contexts:

$v ::=$	values
$\quad \lambda x{:}A.e$	$\quad$ function
$C ::=$	evaluation contexts
$\quad \bullet$	$\quad$ hole
$\quad e\ C$	$\quad$ application right
$\quad C\ v$	$\quad$ application left

and the reduction rules:

$$C[(\lambda x{:}A.e\ v)] \to C[e[v/x]]$$
$$C[\langle e : A \text{ with } T\rangle] \to C \wr \langle e : A \text{ with } T\rangle[e]$$

The following decomposition lemma is easy.

Lemma 1. *Suppose that e is a closed term. Then e is a value or $e = C[r]$ for some context C and term r which is either a β-redex or an implementation.*

A more general lemma is necessary when we extend the language, and then we want to distinguish arbitrary normal forms (like the bad application 3(4)) from proper values.

Since Theorem 1 applies to any well-typed term in System $\mathcal{E}^*$, it is easy to check:

Theorem 2. *If $\Gamma \vdash C[r] : R$ in System $\mathcal{E}^*$, ε does not appear in Γ, and $C[r] \rightarrow e$, then there exists R' such that $\Gamma \vdash e : R'$ in System $\mathcal{E}^*$.*

The following corollary follows from Lemma 1 and Theorem 2.

Corollary 1. *Suppose that e is a closed term. If e is well-typed in System $\mathcal{E}^*$ then either e is a value or there exists e' such that $e \rightarrow e'$ and e' is also well-typed in System $\mathcal{E}^*$.*

4.3 A Programming Syntax

We believe that System $\mathcal{E}^*$ suggests useful (and not too esoteric) ideas for language design. Even if realizing these ideas is beyond the aims of this paper, we can already outline a more practical syntax for choice.

In actual programming, one often wants to attach names to types for conciseness and clarity, and also as brands (in the sense of Modula-3 [20]), in order to distinguish different uses of the same type (for instance, real temperatures and real distances). We use type variables as names for ε types. When X is associated with A in the typing context, X stands for $\varepsilon X.A$. When X and Y are associated with A and $A[Y/X]$, respectively, we need not equate them.

We further permit simultaneous choices, so that an interface does not have to rely on one single type variable. Thus, for example, we may write an interface type of address books, with several type variables for the types of names, phone numbers, and addresses. Simultaneous choices are convenient but not a major extension. (We believe that they can be reduced to simple choices using quantifiers.)

The typing judgements are of the form $\Delta, \Gamma \vdash e : T$ where Γ binds term variables to types as usual, and Δ is a list of associations, each of the form $[X_1, \ldots, X_n | A]$. The grammar of the language is:

$A, B, T ::=$	types
$\quad X, Y, \ldots$	$\quad$ type variables
$\quad A \rightarrow B$	$\quad$ function type
$e, t, u ::=$	terms
$\quad x$	$\quad$ variable
$\quad \lambda x{:}A.t$	$\quad$ function
$\quad t\ u$	$\quad$ application
$\quad \langle t \mid X_1 = T_1 \ldots X_n = T_n \rangle$	$\quad$ implementation

```
List(L) ≡ {nil : L; cons : Nat → L → L; hd : L → Nat option; tl : L → L}
let bl = {nil = []; cons = (_ :: _);                                  : List(Nat list)
          hd = function (a :: l) → Some a | _ → none;
          tl = function (a :: l) → l | _ → []}
in let cons2 x y a =                  : Nat → Nat → List(εL.List(L)) → List(εL.List(L))
   let l = ⟨bl : List(L) with L = Nat list⟩ in l.cons x (l.cons y a)
in let hd2 a =                                  : List(εL.List(L)) → (Nat ∗ Nat) option
   let l = ⟨bl : List(L) with L = Nat list⟩ in
      match (l.hd a, l.hd(l.tl a) with
        Some x, Some y → Some(x, y)
      | _ → none
in let tl2 = ...                                : List(εL.List(L)) → List(εL.List(L))
in let l = ⟨{nil = ⟨bl : List(L) with L = Nat list⟩.nil;                : List(εL.List(L))
             cons = fun x a → match hd2 a  with
                                 Some(n, x′) where x = x′ → cons2 (n + 1) x (tl2 a)
                                |_ → cons2 1 x a;
                  ...} with L = εL.List(L)⟩
in(cons2 3 3 l.nil)                                                   : List(εL.List(L))
```

Fig. 1. An incremental example

Pleasantly, the restriction to ε^* is embedded in the syntax. We need a well-formedness condition for the associations Δ:

Definition 3. *The list of associations Δ is well-formed if it is empty or is of the form $\Delta'; [X_1, \ldots, X_n|A]$ with Δ' well-formed, $X_1, \ldots, X_n$ pairwise distinct and not already bound in Δ', and every type variable that occurs in A being either some X_i or bound in Δ'. Further, Δ, Γ is well-formed if Δ is well-formed and all type variables used in Γ are bound in Δ.*

The typing rules for judgements of the form $\Delta, \Gamma \vdash e : T$ are the ones of the simply typed λ-calculus with the provision that Δ, Γ is well-formed. The typing rule for implementations is:

$$\frac{\Delta, \Gamma \vdash e : A[T_1/X_1, \ldots, T_n/X_n] \quad [X_1, \ldots X_n|A] \in \Delta}{\Delta, \Gamma \vdash \langle e|X_1 = T_1 \ldots X_n = T_n\rangle : A}$$

Here $[T_1/X_1, \ldots, T_n/X_n]$ represents parallel substitution.

We can adapt the computation rules to this syntax, but the specifics are somewhat complicated. We detail only a simple case in order to show how branding becomes apparent. Provided $X_1, \ldots, X_n$ do not appear in the remaining of the typing context, we have:

$$\begin{array}{l} C[\langle e|X_1 = T_1 \ldots X_n = T_n\rangle] \quad \rightarrow \\ \qquad C[T_1/X_1, \ldots, T_n/X_n, e/\langle\star|X_1 = \star \ldots X_n = \star\rangle][e] \end{array}$$

An important aspect of this rule is that linking of implementations no longer require any inefficient testing of type equalities.

5 An Incremental Example

We can now study an example in more detail. Its code is given in Figure 1. For clarity, we have added the types of the main functions on the right-hand side. This example features another simple interface for lists, composed of the empty list, consing, and the head and tail functions. We have adopted an ML syntax (actually close to Caml) and we suppose the language supports primitive lists and pattern matching. The primitive empty list (respective primitive consing) is written $[]$ (respectively $a :: l$). We also use records and an option type, both in a standard way. The record of the interface is abbreviated as $\mathsf{List}(L)$. The type of lists is thus $\varepsilon L.\mathsf{List}(L)$.

The example defines two possible instantiations for the type $\mathsf{List}(\varepsilon L.\mathsf{List}(L))$. The first trivially packages the primitive implementation. The second is optimized for lists with many identical successive elements: for instance, $[1;1;1;4;4]$ is represented by $[3;1;2;4]$.

Interestingly, the second implementation is built upon the first one. Thus $\varepsilon L.\mathsf{List}(L)$ is linked successively to the two implementations during execution. Furthermore, the function cons2 is used in the definition of the optimized representation, and will thus behave in two different ways. This illustrates the possibility of using System $\mathcal{E}^*$ for a form of incremental programming.

The final result of the evaluation is $[2;3]$, which is the "optimized" representation of $[3;3]$. In order to describe the evaluation concisely, we need some abbreviations. Let c_2, b, and bl be the terms such that the program reads:

$\mathsf{List}(L) \equiv \{\mathsf{nil} : L; \dots\}$
let bl $= bl$
in let cons2 $= c_2$
in let hd2 $= \dots$
in let tl2 $= \dots$
in let $l = b$ in(cons2 3 3 l.nil)

No linking takes place during the first steps of the evaluation; instead the four first let constructs are substituted, so that the program is then of the form:

$$(\mathsf{let}\ l = b\ \mathsf{in}\ (\mathsf{cons2}\ 3\ 3\ l.\mathsf{nil}))[\sigma]$$

where σ stands for the successive substitutions $[bl/\mathsf{bl}] \circ [c_2/\mathsf{cons2}] \dots$

Then comes the evaluation of $b[\sigma]$ which yields a first linking step:

$$(\mathsf{let}\ l = b\ \mathsf{in}\ (\mathsf{cons2}\ 3\ 3\ l.\mathsf{nil}))[\sigma] \wr b[\sigma]$$

which is equal to:

$$\mathsf{let}\ l = b[\sigma] \wr b[\sigma]\ \mathsf{in}\ ((\mathsf{cons2}\ 3\ 3\ l.\mathsf{nil}) \wr b)[\sigma]$$

However $b[\sigma] \wr b[\sigma]$ is a record and the evaluation of its nil component involves a *second* linking; further reducing the let construct and the arguments of the function we reach:

$$((\mathsf{cons2}\ 3\ 3\ [])[\sigma] \wr b[\sigma]) \wr \langle bl : \mathsf{List}(L)\ \mathsf{with}\ L = \mathsf{Nat\ list}\rangle$$

which is actually equal to:

$$(c_2[bl/\mathsf{bl}] \wr b[\sigma]) \wr (\langle bl : \mathsf{List}(L) \text{ with } L = \mathsf{Nat\ list}\rangle)\ 3\ 3\ []$$

Here, we see precisely that in the body of the first occurrence of cons2, lists are linked to the "optimized" implementation by $b[\sigma]$. However inside the body of $b[\sigma]$, lists are linked to the standard implementation by $\langle bl : \mathsf{List}(L) \text{ with } L = \mathsf{Nat\ list}\rangle$. As a result, the function cons2 has two different behaviors, depending upon whether it is used inside or outside b.

This example thus suggests that, in this setting, functions cannot simply be viewed as closures. It also raises the question of an efficient execution model for System $\mathcal{E}^*$. We omit our answer to that question.

6 Parametricity, Termination, Conservativity (Sketch)

We close the technical material of this paper with brief discussions of parametricity, of normalization, and of the logical strength of ε.

6.1 Conflict with Parametricity

System F is parametric in the sense that computations do not depend on type information. On the other, System $\mathcal{E}$ and System $\mathcal{E}^*$ clearly lack parametricity. Specifically, we can find terms $e[T/X]$ and $e[T'/X]$ that yield different outputs:

- Let e be the term

$$\langle \lambda x{:}X.x : X \to X\rangle; \langle \lambda x{:}\mathsf{Int}.0 : \mathsf{Int} \to \mathsf{Int}\rangle(1)$$

  (This term is well-typed as soon as we allow the use of type variables at all.)
- $e[\mathsf{Int}/X]$ yields 1.
- $e[\mathsf{Bool}/X]$ yields 0.

We have yet to investigate semantic models for calculi with ε. The failure of parametricity suggests that such models might be rather different from the models of System F.

6.2 Conflict with Normalization

In extensions of System F, non-parametricity often conflicts with strong normalization. In particular, Girard studied a non-parametric combinator J and showed that its addition to System F breaks strong normalization [9]. Harper and Mitchell considered a related combinator J' which also breaks strong normalization despite having a more mundane type than J [11]. Both J and J' rely on testing type equalities at run-time.

Using similar ideas, we can exhibit a non-terminating term which is well-typed in System $\mathcal{E}$ with impredicative universal quantification. For brevity, we

do not detail the standard rules for universal types here, and we do not discuss which restrictions can preserve type soundness in the resulting calculus since our example presents no problem in that respect.

Consider the following abbreviations:

$$\begin{array}{ll} \mathsf{B} \equiv \forall X.X \to X \to X & \text{(the type of booleans encoded in System F)} \\ m_1 = \langle \lambda x{:}X.\lambda f{:}\mathsf{B}.(f\ \mathsf{B}\ f\ f) : X \to \mathsf{B} \to \mathsf{B} \rangle & q = m_1; m_2 : X \to X \to X \\ m_2 = \langle \lambda x{:}X.\lambda y{:}X.x : X \to X \to X \rangle & r = \Lambda X.q : \mathsf{B} \end{array}$$

Since $r \wr m_1[\mathsf{B}/X] = r$ and $q[\mathsf{B}/X] \wr m_1[\mathsf{B}/X] = \lambda x{:}\mathsf{B}.\lambda f{:}\mathsf{B}.(f\ \mathsf{B}\ f\ f)$, the well-typed term $(r\ \mathsf{B}\ r\ r)$ reduces to itself in any weak call-by-value reduction strategy.

An analogous term can be written without quantifiers or in programming syntax, but then we recover termination. The main open question on this subject is whether every well-typed term of System $\mathcal{E}^*$ terminates with our operational semantics. The evidence thus far suggests that, if true, termination might be tricky to justify.

6.3 Non-conservativity

The addition of ε is not conservative over intuitionistic second-order propositional logic. Specifically, we can derive $\exists Z.((\exists Z.A) \to A[Z])$ using ε, but not otherwise. This property suggests that computation mechanisms beyond those present in intuitionistic systems might in general be necessary for programming languages with ε (much as happens for calculi based on classical logic).

On the other hand, $\exists Z.((\exists Z.A) \to A[Z])$ is essentially all we get. We have obtained a compositional translation from our λ-calculus with ε to second-order λ-calculus, which yields a term parameterized by variables of type $\exists Z.((\exists A) \to A[Z])$, for each εA implemented in the term. The type $\exists Z.((\exists A) \to A[Z])$ is a close relative of $\exists Z.(A[Z] \to (\forall A))$, the drinker's paradox ("there exists Z such that if Z drinks then everyone drinks"). The details of this translation, which extends to systems with quantifiers, are however beyond the scope of this paper.

In particular, ε does not yield the full power of classical logic. We cannot derive the classical drinker's paradox, or the law of the excluded middle. (The operator ε can be interpreted over a little 3-point Heyting algebra in which the law of the excluded middle fails.)

7 Related and Further Work

The literature contains much material on the choice operator, on abstract datatypes, and on linking. Of course this material includes Hilbert's original work. It also includes many more recent—and sometimes more exotic—developments. For instance, ε has been used in explaining natural-language quantifiers. In what follows, we discuss the research most closely related to ours.

On the logical side, the most relevant research is that on the proof theory of logical systems with choice. In particular, Leisenring studied cut elimination

in a classical first-order logic with choice. Leisenring defined an order for resolving choices that depended on a delicate system of ranks. Flannagan later discovered and corrected a flaw in Leisenring's definition [8]. Mints also studied the proof theory of choice, in particular for intuitionistic systems [16] (see also [17]). Leivant considered choice in intuitionistic arithmetic, where the addition of choice quickly leads to a classical system [15]. Some other prior work is described for example in a recent encyclopedia article [1]. A general characteristic of this work is that it relies on proof transformations with sensible technical motivations but which do not necessarily correspond to sensible evaluation strategies from a computing perspective, as indicated in section 3.

On the computing side, choice is related to many familiar constructs and phenomena—such as dynamic linking, as we argue in this paper, and others discussed in the introduction. We are not however aware of any programming-language treatment of the choice operator. Recently, in intriguing unpublished work, J.-L. Krivine has been exploring the computational meaning of certain classical-logic formulas that express choice principles [13]. Formally, our systems are quite different, and Krivine tends to explain those formulas in terms of object-oriented programming (rather than linking). Despite such differences, we owe much to Krivine's set-theoretic investigations in the late 1990s.

Considering that the choice operator can often be seen as syntactic sugar, one might imagine that proof theory and type theory would hardly be affected by the introduction of this operator. One might at least expect to transfer results from systems with quantifiers. This point of view is unfortunately simplistic. In particular, $\varepsilon X.A$ can be encoded as $A[\text{true}/X]$ or as $\neg A[\text{false}/X]$ in classical propositional logic [18]; these trivial encodings preserve provability, but they do not capture computational behavior, and they hardly give us new type systems.

Focusing on computation and types, this paper defines and studies a programming calculus with a choice operator. This investigation suggests much further work. This work includes syntax exploration as hinted in section 4.3 but also extension with other familiar programming-language constructs, such as subtyping, recursion, and mutable references. Another interesting track is to understand how the use of choice may be liberalized, relaxing restrictions adopted in this paper. Finally, it would be worthwhile to reconsider the role of choice in the context of mainstream programming systems—both explaining present systems in logical terms and enriching those systems.

Acknowledgments. Georges Gonthier's work was partly done at INRIA Rocquencourt. Martín Abadi's work was started while at Bell Labs; it was partly supported by the National Science Foundation under Grant CCR-0204162.

References

1. Jeremy Avigad and Richard Zach. The epsilon calculus. In the Stanford Encyclopedia of Philosophy, on the web at `http://plato.stanford.edu/entries/epsilon-calculus`, version of May 3, 2002.

2. Luca Cardelli. Program fragments, linking, and modularization. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277, 15–17 January 1997.
3. Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, Boca Raton, FL, 1997.
4. Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. In *Proceedings of the IFIP TC2 Working Conference on Programming Concepts and Methods*, pages 479–504. North Holland, 1990.
5. Drew Dean. The security of static typing with dynamic linking. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, pages 18–27, 1997.
6. Sophia Drossopoulou, Susan Eisenbach, and David Wragg. A fragment calculus—towards a model of separate compilation, linking and binary compatibility. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, pages 147–156, 1999.
7. Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible models for dynamic linking. In P. Degano, editor, *Proceedings of the 12th European Symposium on Programming, ESOP 2003*, volume 2618 of *LNCS*, pages 38–53. Springer, 2003.
8. T. B. Flannagan. On an extension of Hilbert's second ε-theorem. *Journal of Symbolic Logic*, 40(3):393–397, September 1975.
9. Jean-Yves Girard. *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII, June 1972.
10. Neal Glew and Greg Morrisett. Type safe linking and modular assembly language. In *Conference Record of POPL '99: 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 250–261, January 1999.
11. Robert Harper and John C. Mitchell. Parametricity and variants of Girard's J operator. *Information Processing Letters*, 70(1):1–5, April 1999.
12. Michael Hicks, Stephanie Weirich, and Karl Crary. Safe and flexible dynamic linking of native code. In R. Harper, editor, *Types in Compilation: Third International Workshop, TIC 2000; Montreal, Canada, September 21, 2000; Revised Selected Papers*, volume 2071 of *Lecture Notes in Computer Science*, pages 147–176. Springer, 2001.
13. Jean-Louis Krivine. Two talks about specifications and objects. Lectures at TYPES Spring School, March 2002.
14. A. C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol*. Gordon and Breach Science Publishers, New York, 1969.
15. Daniel Leivant. Existential instantiation in a system of natural deduction for intuitionistic arithmetics. Technical report, Stichting Mathematisch Centrum. Note ZW 13/73.
16. Grigori Mints. Heyting predicate calculus with epsilon-symbol. *Journal of Soviet Mathematics*, 8:317–323, 1977. Preprint provided by the author with an indication that the paper appeared in *Selected Papers in Proof Theory*.
17. Grigori Mints. Strong termination for the epsilon substitution method. *Journal of Symbolic Logic*, 61(4):1193–1205, December 1996.
18. Grigori Mints. Private communication. 2000.
19. John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 37–51, 1985.
20. Greg Nelson, editor. *Systems Programming in Modula-3*. Prentice Hall, 1991.