Dynamic Typing in a Statically Typed Language*

Martín Abadi[†] Luca Cardelli[†] Benjamin Pierce[‡] Gordon Plotkin[§]

Abstract

Statically typed programming languages allow earlier error checking, better enforcement of disciplined programming styles, and generation of more efficient object code than languages where all type consistency checks are performed at run time. However, even in statically typed languages, there is often the need to deal with data whose type cannot be determined at compile time. To handle such situations safely, we propose to add a type `Dynamic` whose values are pairs of a value `v` and a type tag `T` where `v` has the type denoted by `T`. Instances of `Dynamic` are built with an explicit tagging construct and inspected with a type safe `typecase` construct.

This paper explores the syntax, operational semantics, and denotational semantics of a simple language including the type `Dynamic`. We give examples of how dynamically typed values can be used in programming. Then we discuss an operational semantics for our language and obtain a soundness theorem. We present two formulations of the denotational semantics of this language and relate them to the operational semantics. Finally, we consider the implications of polymorphism and some implementation issues.

1 Introduction

Statically typed programming languages allow earlier error checking, better enforcement of disciplined programming styles, and generation of more efficient object code than languages where all type consistency checks are performed at run time. However, even in statically typed languages, there is often the need to deal with data whose type cannot be determined at compile time. For example, full static typechecking of programs that exchange data with

---

[*] An earlier version of this paper was published as SRC Research Report #47, copyright Digital Equipment Corporation 1989. A condensed version was presented at the Sixteenth Annual ACM Symposium on Principles of Programming Languages, January, 1989, in Austin, Texas. This extended version appeared in ACM Transactions on Programming Languages and Systems (volume 13, number 2, pp. 237–268, April 1991), and is here published by permission.

[†] Digital Equipment Corporation, Systems Research Center.

[‡] School of Computer Science, Carnegie Mellon University. (This work was started at DEC Systems Research Center.)

[§] Department of Computer Science, University of Edinburgh. (This work was started at Stanford's Center for the Study of Language and Information.)

other programs or access persistent data is in general not possible. A certain amount of dynamic checking must be performed in order to preserve type safety.

Consider a program that reads a bitmap from a file and displays it on a screen. Probably the simplest way to do this is to store the bitmap externally as an exact binary image of its representation in memory. (For concreteness, assume that the bitmap is stored internally as a pair of integers followed by a rectangular array of booleans.) But if we take strong typing seriously, this is unacceptable: when the data in the file happens *not* to be two integers followed by a bit string of the appropriate length, the result can be chaos. The safety provided by static typing has been compromised.

A better solution, also widely used, is to build explicit procedures for reading and writing bitmaps—storing them externally as character strings, say, and generating an exception if the contents of the file are not a legal representation of a bitmap. This amounts essentially to decreeing that there is exactly one data type external to programs and to requiring that all other types be *encoded* as instances of this single type. Strong typing can now be preserved—at the cost of some programming. But as software systems grow to include thousands of data types, each of which must be supplied with printing and reading routines, this approach becomes less and less attractive. What is really needed is a combination of the convenience of the first solution with the safety of the second.

The key to such a solution is the observation that, as far as safety is concerned, the important feature of the second solution is not the details of the encoding of a bitmap as a string, but the fact that it is possible to generate an exception if a given string does not represent a bitmap. This amounts to a run-time check of the type correctness of the `read` operation.

With this insight in hand, we can combine the two solutions above: the contents of a file should include both a binary representation of a data object and a representation of its type. The language can provide a single `read` operation that checks whether the type in the file matches the type declared for the receiving variable. In fact, rather than thinking of files as containing two pieces of information—a data object and its type—we can think of them as containing a *pair* of an object and its type. We introduce a new data type called `Dynamic` whose values are such pairs, and return to the view that all communication with the external world is in terms of objects of a single type—no longer `String`, but `Dynamic`. The `read` routine itself does no run-time checks, but simply returns a `Dynamic`. We provide a language construct, `dynamic` (with a lowercase "`d`"), for packaging a value together with its type into a `Dynamic` (which can then be "externed" to a file), and a `typecase` construct for inspecting the type tag of a given `Dynamic`.

We might use `typecase`, for example, to display the entire contents of a directory where each file may be either a bitmap or a string:

```
foreach filename in openDir("MyDir") do
  let image = read(filename) in
    typecase image of
      (b:Bitmap)
         displayBitmap(b)
      (s:String)
         displayString(s)
```

```
          else
             displayString("<???>")
        end
```

This example can be generalized by making the directory itself into a `Dynamic`. Indeed, the entire file system could be based on `Dynamic` structures. `Dynamic` objects can also be used as the values exchanged during interprocess communication, thereby providing type safe interactions between processes. The Remote Procedure Call paradigm [4] uses essentially this mechanism. (Most RPC implementations optimize the conversions to and from the transport medium, so the `Dynamic` objects may exist only in principle.)

A number of systems already incorporate mechanisms similar to those we have described. But so far these features have appeared in the context of full-scale language designs, and seldom with a precise formal description of their meaning. No attention has been given to the more formal implications of dynamic typing, such as the problems of proving soundness and constructing models for languages with `Dynamic`.

The purpose of this paper is to study the type `Dynamic` in isolation, from several angles. Section 2 reviews the history of dynamic typing in statically typed languages and describes some work related to ours. Section 3 introduces our version of the `dynamic` and `typecase` constructs and gives examples of programs that can be written with them. Section 4 presents an operational semantics for our language and obtains a syntactic soundness theorem. Section 5 investigates two semantic models for the same language and their relation to the operational semantics. Section 6 outlines some preliminary work on extending our theory to a polymorphic lambda-calculus with `Dynamic`. Section 7 discusses some of the issues involved in implementing `Dynamic` efficiently.

2 History and Related Work

Since at least the mid-1960s, a number of languages have included finite disjoint unions (e.g. Algol-68) or tagged variant records (e.g. Pascal). Both of these can be thought of as "finite versions" of `Dynamic`: they allow values of different types to be manipulated uniformly as elements of a tagged variant type, with the restriction that the set of variants must be fixed in advance. Simula-67's subclass structure [5], on the other hand, can be thought of as an infinite disjoint union—essentially equivalent to `Dynamic`. The Simula-67 `INSPECT` statement allows a program to determine at run time which subclass a value belongs to, with an `ELSE` clause for subclasses that the program doesn't know or care about.

CLU [20] is a later language that incorporates the idea of dynamic typing in a static context. It has a type `any` and a `force` construct that attempts to coerce an `any` into an instance of a given type, raising an exception if the coercion is not possible. Cedar/Mesa [19] provides very similar `REFANY` and `TYPECASE`. These features of Cedar/Mesa were carried over directly into Modula-2+ [33] and Modula-3 [8, 9]. In CLU and Cedar/Mesa, the primary motivation for including a dynamic type was to support programming idioms from LISP.

Shaffert and Scheifler gave a formal definition [34] and denotational semantics [35] of CLU, including the type `any` and the `force` construct. This semantics relies on a domain of run time values where *every* value is tagged with its compile time type. Thus, the coercion mapping a value of a known type into a value of type `any` is an identity function; `force` can always look at a value and read off its type. Our approach is more refined, since it distinguishes those values whose types may need to be examined at run time from those that can be stripped during compilation. Moreover, the semantic definition of CLU has apparently never been proved to be sound. In particular, it is not claimed that run time values actually occurring in the evaluation of a well-typed program are tagged with the types that they actually possess. The proof of a soundness result for CLU would probably require techniques similar to those developed in this paper.

ML [16, 25, 26] and its relatives have shown more resistance to the incorporation of dynamic typing than languages in the Algol family. Probably this is because many of the uses of `Dynamic` in Algol-like languages are captured in ML by polymorphic types. Moreover, until recently ML has not been used for building software systems that deal much with persistent data. Still, there have been various proposals for extending ML with a dynamic type. Gordon seems to have thought of it first [15]; his ideas were later extended by Mycroft [28]. The innovation of allowing pattern variables in `typecase` expressions (see below) seems to originate with Mycroft. (Unfortunately, neither of these proposals were published.) Recent versions of the CAML language [38] include features quite similar to our `dynamic` and `typecase` constructs.

Amber [7], a language based on subtyping, includes a `Dynamic` type whose main use is for handling persistent data. In fact, the Amber system itself depends heavily on dynamically typed values. For example, when a module is compiled, it is stored in the file system as a single `Dynamic` object. Uniform use of `Dynamic` in such situations greatly simplifies Amber's implementation.

The use of dynamically typed values for dealing with persistent data seems to be gaining in importance. Besides Amber, the mechanism is used pervasively in the Modula-2+ programming environment. A `REFANY` structure can be "pickled" into a bytestring or a file, "unpickled" later by another program, and inspected with `TYPECASE`. Dynamically typed objects have also been discussed recently in the database literature as an approach to dealing with persistent data in the context of statically typed database programming languages [1, 2, 10].

Recently, Thatte [36] has described a "quasi-static" type system based on the one described here, where our explicit `dynamic` and `typecase` constructs are replaced by implicit coercions and run time checks.

3 Programming with Dynamic

This section introduces the notation used in the rest of the paper—essentially Church's simply typed lambda-calculus [12, 17] with a call-by-value reduction scheme [30], extended with the type `Dynamic` and the `dynamic` and `typecase` constructs. We present a number of example programs to establish the notation and illustrate its expressiveness.

Our fundamental constructs are λ-abstraction, application, conditionals, and arithmetic on natural numbers. We write `e` $\Rightarrow$ `v` to show that an expression `e` evaluates to a value `v`, and `e:T` to show that an expression `e` has type `T`. For example,

```
+ : Nat→Nat→Nat
5+3 ⇒ 8
(λf:Nat→Nat.f(0)) : (Nat→Nat)→Nat
(λf:Nat→Nat.f(0)) (λx:Nat.x+1) ⇒ 1
```

In order to be able to consider evaluation and typechecking separately, we define the behavior of our evaluator over all terms—not just over well-typed terms. (In a compiler for this language, the typechecking phase might strip away type annotations before passing programs to an interpreter or code generator. Our simple evaluator just ignores the type annotations.)

Of course, evaluation of arbitrary terms may encounter run-time type errors such as trying to apply a number as if it were a function. The result of such computations is the distinguished value `wrong`:

```
(5 6)             ⇒  wrong
(λz:Nat.0) (5 6)  ⇒  wrong
```

Note that in the second example a run-time error occurs even though the argument `z` is never used in `(λz.0)`: we evaluate expressions in applicative order. Also, note that `wrong` is different from ⊥ (nontermination). This allows us to distinguish in the semantics between programs that loop forever, which may be perfectly well typed, and programs that crash because of run-time type errors.

To make the examples in this section more interesting we also use strings, booleans, cartesian products, and recursive λ-expressions, all of which are omitted in the formal parts of the paper. Strings are written in double quotes; || is the concatenation operator on strings. Binary cartesian products are written with angle brackets; `fst` and `snd` are projection functions returning the first and second components of a pair. Recursive lambda expressions are written using the fixpoint operator `rec`, where we intend `rec(f:U→T)λx:U.e` to denote the least-defined function `f` such that, informally, `f` = `λx:U.e`. For example,

```
<λx:Nat.x+1,1> : (Nat→Nat)×Nat
snd(<λx:Nat.x+1,1>) ⇒ 1

(rec(f:Nat→Nat)λn:Nat.
    if n=0 then 1 else n*f(n-1)) (5)
⇒ 120
```

We show at the end of this section that recursive λ-expressions actually need not be primitives of the language: they can be defined using `Dynamic`.

Values of type `Dynamic` are built with the `dynamic` construct. The result of evaluating the expression `dynamic e:T` is a pair of a value `v` and a type tag `T`, where `v` is the result of evaluating `e`. The expression `dynamic e:T` has type `Dynamic` if `e` has type `T`.

The `typecase` construct is used to examine the type tag of a `Dynamic` value. For example, the expression

```
λx:Dynamic.
    typecase x of
       (i:Nat)  i+1
```

```
      else    0
   end
```

applied to `(dynamic 1:Nat)`, evaluates to `2`. The evaluator attempts to match the type tag of `x` against the pattern `Nat`, succeeds, binds `i` to the value component of `x`, adds `1` to `i`, and returns the result.

The patterns in the case branches need not fully specify the type they are to match: they may include "pattern variables," which match any subexpression in the type tag of the selector. The pattern variables are listed between parentheses at the beginning of each guard, indicating that they are bound in the branch.

The full syntax of `typecase` is

```
typecase e_sel of
   ...
   (X⃗_i) (x_i:T_i) e_i
   ...
   else e_else
end
```

where e_{sel}, e_{else}, and e_i are expressions, x_i are variables, T_i are type expressions, and $\vec{X}_i$ are lists of distinct type variables. (It will sometimes be convenient to treat the $\vec{X}_i$ as a set rather than a list.) If any of the $\vec{X}_i$ are empty, their enclosing parentheses may be omitted. The occurrences of type variables in X_i are binding and have scope over the whole branch, that is, over both T_i and e_i.

If the type tag of a typecase selector matches more than one guard, the first matching branch is executed. There are other possible choices here. For instance, we could imagine requiring that the patterns form an "exclusive and exhaustive" covering of the space of type expressions so that a given type tag always matches exactly one pattern [28].

One example using dynamic values is a function that returns a printable string representation of any dynamic value:

```
rec(tostring: Dynamic→String)
  λdv:Dynamic.
    typecase dv of
       (v: String) "\"" || v || "\""
       (v: Nat) natToStr(v)
       (X,Y) (v: X→Y) "<function>"
       (X,Y) (v: X×Y)
          "<" || tostring(dynamic fst(v):X) || ","
              || tostring(dynamic snd(v):Y) || ">"
       (v: Dynamic)
          "dynamic " || tostring(v)
       else "<unknown>"
     end
```

The case for pairs illustrates a subtle point. It uses a pattern to match *any* pair, and then calls the `tostring` function recursively to convert the components. To do this, it must package them into new dynamic values by tagging them with their types. This is possible because the variables `X` and `Y` are bound at run time to the appropriate types.

Since the type tag is part of the run-time representation of a dynamic value, the case for `Dynamic` should probably return a string representation not only of the tagged value but of the type tag itself. This is straightforward, using an auxiliary function `typetostring` with the same structure as `tostring`.

```
rec(typetostring: Dynamic→String)
  λdv:Dynamic.
    typecase dv of
      (v: String) "String"
      (v: Nat) "Nat"
      (X,Y) (v: X→Y) "<function>"
      (X,Y) (v: X×Y)
          typetostring(dynamic fst(v):X)
          || "×"
          || typetostring(dynamic snd(v):Y)
      (v: Dynamic) "Dynamic"
      else "<unknown>"
    end
```

Neither `tostring` nor `typetostring` quite do their jobs: for example, when `tostring` gets to a function, it stops without giving any more information about the function. It can do no better, given the mechanisms we have described, since there is no effective way to get from a function value to an element of its domain or codomain. This limitation even precludes using `typetostring` to show the domain and codomain types of the function, since the argument to `typetostring` must be a value, not just a disembodied type.

It would be possible to add another mechanism to the language, providing a way of "unpackaging" the type tag of a `Dynamic` into a data structure that could then be examined by the program. (Amber [7] and Cedar/Mesa [19] have this feature.) Although this would be a convenient way to implement operations like type printing—which may be important in practice—we believe that most of the theoretical interest of `Dynamic` lies in the interaction between statically and dynamically checked parts of the language that the `typecase` expression allows. Under the proposed extension, a function could behave differently depending on the type tag of a dynamic value passed as a parameter, but the *type* of its result could not be affected without giving up static typechecking.

Another example, demonstrating the use of nested typecase expressions, is a function that applies its first argument to its second argument, after checking that the application is correctly typed. Both arguments are passed as dynamic values, and the result is a new dynamic value. When the application fails, the type tag of the result will be `String` and its value part will be `"Error"`. (In a richer language we could raise an exception in this case.)

```
λdf:Dynamic. λde:Dynamic.
  typecase df of
    (X,Y) (f: X → Y)
        typecase de of
```

```
            (e: X) dynamic f(e):Y
            else dynamic "Error":String
          end
      else dynamic "Error":String
    end
```

Note that in the first guard of the inner typecase, `X` is not listed as a bound pattern variable. It is not intended to match any type whatsoever, but only the domain type of `f`. Therefore, it retains its binding from the outer pattern, making it a constant as far as the inner typecase is concerned.

Readers may enjoy the exercise of defining a similar function that takes two functions as dynamic values and returns their composition as a dynamic value.

In contrast to some languages with features similar to `Dynamic` (for example, Modula-2+ [33]), the set of type tags involved in a computation cannot be computed statically: our `dynamic` expressions can cause the creation of new tags at run time. A simple example of this is a function that takes a dynamic value and returns a `Dynamic` whose value part is a pair, both of whose components are equal to the value part of the original dynamic value:

```
    λdx:Dynamic.
        typecase dx of
            (X) (x: X)
                dynamic <x,x>: X × X
            else dx
        end
```

It is easy to see that the type tag on the dynamic value returned by this function must be constructed at run time, rather than simply being chosen from a finite set of tags generated by the compiler.

Our last application of `Dynamic` is more substantial. We show that it can be used to build a fixpoint operator, allowing recursive computations to be expressed in the language even without the `rec` construct. It is well known that fixpoint operators cannot be expressed in the ordinary simply typed lambda-calculus. (This follows from the strong normalization property [17, p. 163].) However, by hiding a certain parameter inside a dynamic value, smuggling it past the type system, and unpackaging it again where it is needed, we can write a well-typed version in our language.

A fixpoint of a function `f` is an argument `x` for which `f(x) = x` (our use of the equality sign here is informal). A fixpoint operator `fix` is a function that returns a fixpoint of a function `f` when applied to `f`:

```
    fix f = f (fix f).
```

In call-by-value lambda-calculi, an extensional version of this property must be used instead: for any argument `a`,

```
    (fix f) a = f(fix f) a
```

One function with this property (a call-by-value version of the standard **Y** combinator [3, p. 131], [30]) can be expressed in an *untyped* variant of our notation by:

```
fix = λf. d d
```

where

```
d = λx. λz. (f (x x)) z.
```

To see that `(fix f) a` = `f (fix f) a` for any function `f` and argument `a`, we calculate as follows.

```
(fix f) a  =  ((λf. d d) f) a
           =  (d d) a
           =  (λz. (f (d d)) z) a
           =  (f (d d)) a
           =  (f ((λf. d d) f)) a
           =  (f (fix f)) a
```

To build something similar in the typed language, we need to do a bit more work. Rather than a single fixpoint function, we have to build a family of functions (one for each arrow type). That is, for each arrow type `T→U` we define a function $\mathtt{fix}_{\mathrm{T}\to\mathrm{U}}$ whose type is `((T→U)→(T→U))→(T→U)`. Unfortunately, there is no way to obtain $\mathtt{fix}_{\mathrm{T}\to\mathrm{U}}$ by just filling in suitable type declarations in the untyped `fix` given above. We need to build it in a more roundabout way.

First, we need an expression $\mathtt{a}_{\mathrm{T}}$ for each type `T`. (It does not matter what the expressions are; we need to know only that there is one for every type.) Define:

$$
\begin{aligned}
&\mathtt{a}_{\mathrm{Nat}} \ \mathtt{=}\ \mathtt{0}\\
&\mathtt{a}_{\mathrm{String}} \ \mathtt{=}\ \mathtt{""}\\
&\mathtt{a}_{\mathrm{T}\times\mathrm{U}} \ \mathtt{=}\ \mathtt{<}\mathtt{a}_{\mathrm{T}}\mathtt{,}\mathtt{a}_{\mathrm{U}}\mathtt{>}\\
&\mathtt{a}_{\mathrm{T}\to\mathrm{U}} \ \mathtt{=}\ \lambda\mathtt{x:T.}\ \mathtt{a}_{\mathrm{U}}\\
&\mathtt{a}_{\mathrm{Dynamic}} \ \mathtt{=}\ \mathtt{dynamic\ 0:Nat}
\end{aligned}
$$

Next, we build a family of "embedding" functions from each type `T` into `Dynamic`, and corresponding "projection" functions from `Dynamic` to `T`:

$$
\begin{aligned}
&\mathtt{emb}_{\mathrm{T}} \ \mathtt{=}\ \lambda\mathtt{x:T.\ dynamic\ x:T}\\
&\mathtt{proj}_{\mathrm{T}} \ \mathtt{=}\ \lambda\mathtt{y:Dynamic.}\\
&\qquad\qquad\mathtt{typecase\ y\ of}\\
&\qquad\qquad\quad\mathtt{(z:T)\ z}\\
&\qquad\qquad\quad\mathtt{else}\ \mathtt{a}_{\mathrm{T}}\\
&\qquad\qquad\mathtt{end}
\end{aligned}
$$

It is easy to see that if an expression `e` of type `T` evaluates to some value `v`, then so does $\mathtt{proj}_{\mathrm{T}}\mathtt{(}\mathtt{emb}_{\mathrm{T}}\mathtt{(e))}$.

Now we are ready to construct $\mathtt{fix}_{\mathrm{T}\to\mathrm{U}}$. Abbreviate:

$$
\begin{aligned}
&\mathtt{emb}\ \mathtt{=}\ \mathtt{emb}_{\mathrm{Dynamic}\to(\mathrm{T}\to\mathrm{U})}\\
&\mathtt{proj}\ \mathtt{=}\ \mathtt{proj}_{\mathrm{Dynamic}\to(\mathrm{T}\to\mathrm{U})}\\
&\mathtt{d}\ \mathtt{=}\ \lambda\mathtt{x:Dynamic.}\ \lambda\mathtt{z:T.\ f\ ((proj\ x)\ x)\ z}
\end{aligned}
$$

To see that `d` is well-typed, assume that `f` has type `(T→U)→(T→U)`. The type of `d` works out to be `Dynamic→(T→U)`. Then

$$\mathtt{fix}_{\mathtt{T}\to\mathtt{U}}\ \mathtt{=}\ \lambda\mathtt{f:((T\to\ U)\to\ (T\to\ U)).\ d\ (emb\ d)}$$

has type `((T→U)→(T→U))→(T→U)`, as required, and has the correct behavior.

4 Operational Semantics

We now formally define the syntax of the simply typed lambda-calculus with `Dynamic` and give operational rules for typechecking and evaluation.

4.1 Notation

TVar is a countable set of type variable identifiers. *TExp* is the class of type expressions defined over these by the following BNF equation, where `T` and `U` range over *TExp* and `X` ranges over *TVar*:

$$
\begin{array}{lcl}
\mathtt{T} & ::= & \mathtt{Nat} \\
 & | & \mathtt{X} \\
 & | & \mathtt{T} \to \mathtt{U} \\
 & | & \mathtt{Dynamic}
\end{array}
$$

Similarly, *Var* is a countable set of variables and *OpenExp* is the class of open expressions defined by the following equation, where `e` ranges over *OpenExp*, `x` over *Var*, and `T` over *TExp*:

$$
\begin{array}{lcl}
\mathtt{e} & ::= & \mathtt{x} \\
 & | & \mathtt{wrong} \\
 & | & \lambda\mathtt{x:T.e}_{\mathrm{body}} \\
 & | & \mathtt{e}_{\mathrm{fun}}\mathtt{(e}_{\mathrm{body}}\mathtt{)} \\
 & | & \mathtt{0} \\
 & | & \mathtt{succ\ e}_{\mathrm{nat}} \\
 & | & \mathtt{test\ e}_{\mathrm{nat}}\ \mathtt{0:e}_{\mathrm{zero}}\ \mathtt{succ(x):e}_{\mathrm{succ}} \\
 & | & \mathtt{dynamic\ e}_{\mathrm{body}}\mathtt{:T} \\
 & | & \mathtt{typecase\ e}_{\mathrm{sel}}\ \mathtt{of} \\
 & & \quad \ldots \\
 & & \quad \mathtt{(}\vec{\mathtt{X}}_{\mathrm{i}}\mathtt{)\ (x}_{\mathrm{i}}\mathtt{:T}_{\mathrm{i}}\mathtt{)\ e}_{\mathrm{i}} \\
 & & \quad \ldots \\
 & & \quad \mathtt{else\ e}_{\mathrm{else}} \\
 & & \mathtt{end}
\end{array}
$$

Recall that $\vec{\mathtt{X}}_{\mathrm{i}}$ denotes a list of distinct type variables, and that if the list is empty the enclosing parentheses may be omitted.

This is a simpler language than we used in the examples. We have omitted strings, booleans, cartesian products, and built-in recursive λ-expressions. The natural numbers, our only built-in datatype, are presented by `0`, `succ`, and `test`. The `test` construct helps reduce the low-level clutter in our definitions by subsuming the usual `if...then...else...` construct, test for zero, predecessor function, and boolean datatype into a single construct. It is based on Martin-Löf's elimination rule for natural numbers [23].

We give special names to certain subsets of *TExp* and *OpenExp*. $FTV(\mathtt{e})$ is the set of free type variables in `e`. $FV(\mathtt{e})$ is the set of free variables in `e`. *ClosedExp* denotes the closed expressions; *Exp* denotes the expressions with no free type variables (but possibly with free variables); *TypeCode* denotes the closed type expressions. When we write just "expression," we mean an expression with no free type variables.

Evaluation is taken to be a relation between expressions and expressions (rather than between expressions and some other domain of values). We distinguish a set *Value* $\subset$ *ClosedExp* of expressions "in canonical form." The elements of *Value* are defined inductively: `wrong` is in canonical form; `0`, `succ 0`, `succ(succ 0)`, ... are in canonical form; an expression $(\lambda\mathtt{x}\mathtt{:T.}\mathtt{e}_{\mathtt{body}})$ is in canonical form if it is closed; an expression `dynamic` $\mathtt{e}_{\mathtt{body}}\mathtt{:T}$ is in canonical form if `T` is closed and $\mathtt{e}_{\mathtt{body}}$ is in canonical form and different from `wrong`.

A substitution σ is a finite function from type variables to closed type expressions, written $[\mathtt{X} \leftarrow \mathtt{T}, \mathtt{Y} \leftarrow \mathtt{U}, \ldots]$. *Subst* denotes the set of all substitutions. $\mathit{Subst}_{\vec{\mathtt{X}_i}}$ denotes the set of substitutions whose domain is $\vec{\mathtt{X}_i}$. We use a similar notation for substitution of canonical expressions for free variables in expressions.

A type environment is a finite function from variables to closed type expressions. To denote the modification of a type environment *TE* by a binding of `x` to `T`, we write $TE[\mathtt{x} \leftarrow \mathtt{T}]$. The empty type environment is denoted by $\emptyset$.

We consistently use certain variables to range over particular classes of objects. The metavariables `x`, `y`, and `z` range over variables in the language. (They are also sometimes used as actual variables in program examples.) The metavariable `e` ranges over expressions. Similarly, `X`, `Y`, and `Z` range over type variables and `T`, `U`, `V`, and `W` range over type expressions. The letter σ ranges over substitutions. *TE* ranges over type environments. Finally, `v` and `w` range over canonical expressions.

These definitions and conventions are summarized in Figures 1 and 2.

4.2 Typechecking

Our notation for describing typechecking and evaluation is a form of "structural operational semantics" [31]. The typing and evaluation functions are specified as systems of inference rules; showing that an expression has a given type or reduces to a given value amounts precisely to giving a proof of this fact using the rules. Because the inference rules are similar to those used in systems for natural deduction in logic, this style of description has also come to be known as "natural semantics" [18].

Nat	numbers
Var	variables
TVar	type variables
TExp	type expressions
$\mathit{TypeCode} = \{\mathtt{T} \in \mathit{TExp} \mid \mathit{FTV}(\mathtt{T}) = \emptyset\}$	closed types
$\mathit{OpenExp}$	open expressions
$\mathit{ClosedExp} = \{\mathtt{e} \in \mathit{OpenExp} \mid \mathit{FV}(\mathtt{e}) = \mathit{FTV}(\mathtt{e}) = \emptyset\}$	closed expressions
$\mathit{Exp} = \{\mathtt{e} \in \mathit{OpenExp} \mid \mathit{FTV}(\mathtt{e}) = \emptyset\}$	expressions
$\mathit{Value} = \{\mathtt{e} \in \mathit{OpenExp} \mid \mathtt{e} \text{ in canonical form}\}$	canonical expressions
$\mathit{Subst} = \mathit{TVar} \xrightarrow{\text{fin}} \mathit{TypeCode}$	substitutions
$\mathit{Subst}_{\vec{\mathtt{X}}_{\mathtt{j}}} = \mathit{TVar} \xrightarrow{\text{fin}} \mathit{TypeCode}$	substitutions with domain $\vec{\mathtt{X}}_{\mathtt{j}}$
$\mathit{TEnv} = \mathit{Var} \xrightarrow{\text{fin}} \mathit{TypeCode}$	type environments

Figure 1: Summary of Basic Definitions

$\mathtt{x}$, $\mathtt{y}$, $\mathtt{z}$	variables
$\mathtt{e}$	expressions
$\mathtt{v}$, $\mathtt{w}$	canonical expressions (values)
$\mathtt{X}$, $\mathtt{Y}$, $\mathtt{Z}$	type variables
$\mathtt{T}$, $\mathtt{U}$, $\mathtt{V}$, $\mathtt{W}$	type expressions
σ	substitutions
TE	type environments

Figure 2: Summary of Naming Conventions

The rules closely follow the structure of expressions, and incorporate a strong notion of computation. To compute a type for $\mathtt{e}_{\mathrm{fun}}(\mathtt{e}_{\mathrm{arg}})$, for example, we first attempt to compute types for its subterms $\mathtt{e}_{\mathrm{fun}}$ and $\mathtt{e}_{\mathrm{arg}}$ and then, if we are successful, to combine the results. This exactly mimics the sequence of events we might observe inside a typechecker for the language.

The formalism extends fairly easily to describing a variety of programming language features like assignment statements and exceptions. This breadth of coverage and "operational style" makes the notation a good one for specifying comparatively rich languages like Standard ML [26]. A group at INRIA has built a system for directly interpreting formal specifications written in a similar notation [6, 13, 14].

The rules below define the situations in which the judgement "expression $\mathtt{e}$ has type $\mathtt{T}$" is valid under assumptions TE. This is written "$\mathit{TE} \vdash \mathtt{e} : \mathtt{T}$".

The first rule says that a variable identifier has whatever type is given for it in the type environment. If it is unbound in the present type environment, then the rules simply fail to derive any type. (Technically, the clause "$\mathtt{x} \in \mathit{Dom}(\mathit{TE})$" is not a premise but a side condition that determines when the rule is applicable.)

$$\frac{\mathtt{x} \in \mathit{Dom}(\mathit{TE})}{\mathit{TE} \vdash \mathtt{x} : \mathit{TE}(\mathtt{x})}$$

A λ-expression must have an arrow type. The argument type is given explicitly by the annotation on the bound variable. To compute the result type, we assume that the bound variable has the declared type, and attempt to derive a type for the body under this assumption.

$$\frac{\mathit{TE}[\mathtt{x} \leftarrow \mathtt{U}] \vdash \mathtt{e}_{\mathrm{body}} : \mathtt{T}}{\mathit{TE} \vdash \lambda \mathtt{x}{:}\mathtt{U}.\mathtt{e}_{\mathrm{body}} : (\mathtt{U} {\rightarrow} \mathtt{T})}$$

A well-typed function application must consist of an expression of some arrow type applied to another expression, whose type is the same as the argument type of the first expression.

$$\frac{\begin{array}{c}\mathit{TE} \vdash \mathtt{e}_{\mathrm{fun}} : (\mathtt{U} {\rightarrow} \mathtt{T}) \\ \mathit{TE} \vdash \mathtt{e}_{\mathrm{arg}} : \mathtt{U}\end{array}}{\mathit{TE} \vdash \mathtt{e}_{\mathrm{fun}}(\mathtt{e}_{\mathrm{arg}}) : \mathtt{T}}$$

The constant `0` has type `Nat`.

$$\mathit{TE} \vdash \mathtt{0} : \mathtt{Nat}$$

A `succ` expression has type `Nat` if its body does.

$$\frac{\mathit{TE} \vdash \mathtt{e}_{\mathrm{nat}} : \mathtt{Nat}}{\mathit{TE} \vdash \mathtt{succ}\ \mathtt{e}_{\mathrm{nat}} : \mathtt{Nat}}$$

A `test` expression has type `T` if its selector has type `Nat` and both of its arms have type `T`. The type of the second arm is derived in an environment where the variable `x` has type `Nat`.

$$\frac{\begin{array}{c}\mathit{TE} \vdash \mathtt{e}_{\mathrm{nat}} : \mathtt{Nat} \\ \mathit{TE} \vdash \mathtt{e}_{\mathrm{zero}} : \mathtt{T} \\ \mathit{TE}[\mathtt{x} \leftarrow \mathtt{Nat}] \vdash \mathtt{e}_{\mathrm{succ}} : \mathtt{T}\end{array}}{\mathit{TE} \vdash (\mathtt{test}\ \mathtt{e}_{\mathrm{nat}}\ \mathtt{0}{:}\mathtt{e}_{\mathrm{zero}}\ \mathtt{succ(x)}{:}\mathtt{e}_{\mathrm{succ}}) : \mathtt{T}}$$

A `dynamic` expression is well-typed if the body actually has the type claimed for it.

$$\frac{\mathit{TE} \vdash \mathtt{e}_{\mathrm{body}} : \mathtt{T}}{\mathit{TE} \vdash (\mathtt{dynamic}\ \mathtt{e}_{\mathrm{body}}{:}\mathtt{T}) : \mathtt{Dynamic}}$$

The `typecase` construct is a bit more complicated. In order for an expression of the form $\texttt{(typecase e}_{\mathrm{sel}}\ \texttt{of ...}\ (\vec{\texttt{X}}_{\mathrm{i}})\ (\texttt{x}_{\mathrm{i}}\texttt{:T}_{\mathrm{i}})\ \texttt{e}_{\mathrm{i}}\ \texttt{... end)}$ to have a type `T`, three conditions must be met: First, the selector $\texttt{e}_{\mathrm{sel}}$ must have type `Dynamic`. Second, for every possible substitution σ of typecodes for the pattern variables $\vec{\texttt{X}}_{\mathrm{i}}$, the body $\texttt{e}_{\mathrm{i}}$ of each branch must have type `T`. Third, the `else` arm must also have type `T`.

The second premise is quantified over *all* substitutions $\sigma \in \mathit{Subst}_{\vec{\mathrm{X}}_{\mathrm{i}}}$. Strictly speaking, there are no inference rules that allow us to draw conclusions quantified over an infinite set, so a proof of this premise requires an infinite number of separate derivations. Such infinitary derivations present no theoretical difficulties—in fact, they make the rule system easier to reason about—but a typechecker based naively on these rules would have poor performance. However, our rules can be replaced by a finitary system using skolem constants that derives exactly the same typing judgements.

$$\frac{\begin{array}{c} TE \vdash \texttt{e}_{\mathrm{sel}} : \texttt{Dynamic} \\ \forall i, \forall \sigma \in \mathit{Subst}_{\vec{\mathrm{X}}_{\mathrm{i}}}.\ TE[\texttt{x}_{\mathrm{i}} \leftarrow \texttt{T}_{\mathrm{i}}\sigma] \vdash e_{\mathrm{i}}\sigma : \texttt{T} \\ TE \vdash \texttt{e}_{\mathrm{else}} : \texttt{T} \end{array}}{\begin{array}{l} TE \vdash \texttt{(typecase e}_{\mathrm{sel}}\ \texttt{of} \\ \qquad\quad \texttt{...}(\vec{\texttt{X}}_{\mathrm{i}})\ (\texttt{x}_{\mathrm{i}}\texttt{:T}_{\mathrm{i}})\ \texttt{e}_{\mathrm{i}}\texttt{...} \\ \qquad\quad \texttt{else e}_{\mathrm{else}} \\ \qquad \texttt{end)} \qquad\qquad\quad : \texttt{T} \end{array}}$$

Finally, note that the expression `wrong` is assigned no type. It is the only syntactic form in the language with no associated typing rule.

4.3 Evaluation

The evaluation rules are given in the same notation as the typechecking rules. We define the judgement "closed expression `e` reduces to canonical expression `v`," written "$\texttt{e} \Rightarrow \texttt{v}$," by giving rules for each syntactic construct in the language. In general, there is one rule for the normal case, plus one or two others specifying that the expression reduces to `wrong` under certain conditions.

In this style of semantic description, there is no explicit representation of a nonterminating computation. Whereas in standard denotational semantics an expression that loops forever has the value $\bot$ (bottom), our evaluation rules simply fail to derive any result whatsoever.

When the evaluation of an expression encounters a run-time error like trying to apply a number as if it were a function, the value `wrong` is derived as the expression's value. The evaluation rules preserve `wrong`.

There is no rule for evaluating a variable: evaluation is defined only over closed expressions. Parameter substitution is performed immediately during function application.

The constant `wrong` is in canonical form.

$$\vdash \texttt{wrong} \Rightarrow \texttt{wrong}$$

Every λ-expression is in canonical form.

$$\vdash \lambda\mathtt{x}\mathtt{:}\mathtt{T}\mathtt{.}\mathtt{e}_{\mathrm{body}} \Rightarrow \lambda\mathtt{x}\mathtt{:}\mathtt{T}\mathtt{.}\mathtt{e}_{\mathrm{body}}$$

We have chosen a call-by-value (applicative-order) evaluation strategy: to evaluate a function application, the expression being applied must be reduced to a canonical expression beginning with λ and the argument expression must be reduced to some legal value, that is, its computation must terminate and should not produce `wrong`. If one of these computations results in `wrong`, the application itself reduces immediately to `wrong`. Otherwise the argument is substituted for the parameter variable in the λ body, which is then evaluated under this binding.

$$\frac{\begin{array}{c}\vdash \mathtt{e}_{\mathrm{fun}} \Rightarrow \lambda\mathtt{x}\mathtt{:}\mathtt{T}\mathtt{.}\mathtt{e}_{\mathrm{body}} \\ \vdash \mathtt{e}_{\mathrm{arg}} \Rightarrow \mathtt{w} \quad (\mathtt{w} \neq \mathtt{wrong}) \\ \vdash \mathtt{e}_{\mathrm{body}}[\mathtt{x} \leftarrow \mathtt{w}] \Rightarrow \mathtt{v}\end{array}}{\vdash \mathtt{e}_{\mathrm{fun}}(\mathtt{e}_{\mathrm{arg}}) \Rightarrow \mathtt{v}}$$

$$\frac{\vdash \mathtt{e}_{\mathrm{fun}} \Rightarrow \mathtt{w} \quad (\mathtt{w} \text{ not of the form } (\lambda\mathtt{x}\mathtt{:}\mathtt{T}\mathtt{.}\mathtt{e}_{\mathrm{body}}))}{\vdash \mathtt{e}_{\mathrm{fun}}(\mathtt{e}_{\mathrm{arg}}) \Rightarrow \mathtt{wrong}}$$

$$\frac{\begin{array}{c}\vdash \mathtt{e}_{\mathrm{fun}} \Rightarrow \mathtt{w} \quad (\mathtt{w} = (\lambda\mathtt{x}\mathtt{:}\mathtt{T}\mathtt{.}\mathtt{e}_{\mathrm{body}})) \\ \vdash \mathtt{e}_{\mathrm{arg}} \Rightarrow \mathtt{wrong}\end{array}}{\vdash \mathtt{e}_{\mathrm{fun}}(\mathtt{e}_{\mathrm{arg}}) \Rightarrow \mathtt{wrong}}$$

The constant `0` is in canonical form.

$$\vdash \mathtt{0} \Rightarrow \mathtt{0}$$

A `succ` expression is in canonical form when its body is a canonical number (that is, an expression of the form `0` or `succ n`, where `n` is a canonical number). It is evaluated by attempting to evaluate the body to a canonical value `v`, returning `wrong` if the result is anything but a number and otherwise returning `succ` applied to `v`.

$$\frac{\vdash \mathtt{e}_{\mathrm{nat}} \Rightarrow \mathtt{v} \quad (\mathtt{v} \text{ a canonical number})}{\vdash \mathtt{succ}\ \mathtt{e}_{\mathrm{nat}} \Rightarrow \mathtt{succ}\ \mathtt{v}}$$

$$\frac{\vdash \mathtt{e}_{\mathrm{nat}} \Rightarrow \mathtt{v} \quad (\mathtt{v} \text{ not a canonical number})}{\vdash \mathtt{succ}\ \mathtt{e}_{\mathrm{nat}} \Rightarrow \mathtt{wrong}}$$

A `test` expression is evaluated by evaluating its selector, returning `wrong` if the result is not a number, and otherwise evaluating one or the other of the arms depending on whether the selector is zero or a positive number. In the latter case, the variable `x` is bound inside the arm to the predecessor of the selector.

$$\frac{\begin{array}{c}\vdash \texttt{e}_{\text{nat}} \Rightarrow \texttt{0}\\ \vdash \texttt{e}_{\text{zero}} \Rightarrow \texttt{v}\end{array}}{\vdash \texttt{(test e}_{\text{nat}}\ \texttt{0:e}_{\text{zero}}\ \texttt{succ(x):e}_{\text{succ}}\texttt{)} \Rightarrow \texttt{v}}$$

$$\frac{\begin{array}{c}\vdash \texttt{e}_{\text{nat}} \Rightarrow \texttt{succ w}\\ \vdash \texttt{e}_{\text{succ}}[\texttt{x} \leftarrow \texttt{w}] \Rightarrow \texttt{v}\end{array}}{\vdash \texttt{(test e}_{\text{nat}}\ \texttt{0:e}_{\text{zero}}\ \texttt{succ(x):e}_{\text{succ}}\texttt{)} \Rightarrow \texttt{v}}$$

$$\frac{\vdash \texttt{e}_{\text{nat}} \Rightarrow \texttt{w} \quad (\texttt{w} \text{ not a canonical number})}{\vdash \texttt{(test e}_{\text{nat}}\ \texttt{0:e}_{\text{zero}}\ \texttt{succ(x):e}_{\text{succ}}\texttt{)} \Rightarrow \texttt{wrong}}$$

A `dynamic` expression is evaluated by evaluating its body. If the body reduces to `wrong` then so does the whole `dynamic` expression.

$$\frac{\vdash \texttt{e}_{\text{body}} \Rightarrow \texttt{w} \quad (\texttt{w} \neq \texttt{wrong})}{\vdash \texttt{(dynamic e}_{\text{body}}\texttt{:T)} \Rightarrow \texttt{dynamic w:T}}$$

$$\frac{\vdash \texttt{e}_{\text{body}} \Rightarrow \texttt{wrong}}{\vdash \texttt{(dynamic e}_{\text{body}}\texttt{:T)} \Rightarrow \texttt{wrong}}$$

A `typecase` expression is evaluated by evaluating its selector, returning `wrong` immediately if this produces `wrong` or anything else that is not a dynamic value, and otherwise trying to match the type tag of the selector value against the guards of the typecase. The function *match* has the job of matching a run-time typecode `T` against a pattern expression `U` with free variables. If there is a substitution σ such that $\texttt{T}=\texttt{U}\sigma$, then *match*(`T`, `U`)$=\sigma$. (For the simple type expressions we are dealing with here, σ is unique if it exists.) Otherwise, *match*(`T`, `U`) fails. Section 7.2 discusses the implementation of *match*.

The branches are tried in turn until one is found for which *match* succeeds. The substitution returned by *match* is applied to the body of the branch. Then the selector's value component is substituted for the parameter variable in the body, and the resulting expression is evaluated. (As in the rule for application, we avoid introducing run-time environments by immediately substituting the bound variable $\texttt{x}_{\text{i}}$ and pattern variables $\texttt{T}_{\text{i}}$ into the body of the matching branch.) The result of evaluating the body becomes the value for the whole `typecase`.

If no guard matches the selector tag, the `else` body is evaluated instead.

$$\frac{\begin{array}{c}\vdash \texttt{e}_{\text{sel}} \Rightarrow \texttt{dynamic w:T}\\ \forall j < k.\ \mathit{match}(\texttt{T},\ \texttt{T}_{\text{j}}) \text{ fails}\\ \mathit{match}(\texttt{T},\ \texttt{T}_{\text{k}}) = \sigma\\ \vdash \texttt{e}_{\text{k}}\sigma[\texttt{x}_{\text{k}} \leftarrow \texttt{w}] \Rightarrow \texttt{v}\end{array}}{\begin{array}{l}\vdash \texttt{(typecase e}_{\text{sel}}\ \texttt{of}\\ \qquad \texttt{...(}\vec{\texttt{X}}_{\text{i}}\texttt{) (x}_{\text{i}}\texttt{:T}_{\text{i}}\texttt{) e}_{\text{i}}\texttt{...}\\ \qquad \texttt{else e}_{\text{else}}\texttt{)}\\ \quad \texttt{end)} \qquad\qquad\qquad \Rightarrow \texttt{v}\end{array}}$$

$$\frac{\begin{array}{c}\vdash \mathtt{e_{sel}} \Rightarrow \mathtt{dynamic\ w{:}T} \\ \forall k.\ \mathit{match}(\mathtt{T},\ \mathtt{T_k}) \text{ fails} \\ \vdash \mathtt{e_{else}} \Rightarrow \mathtt{v}\end{array}}{\begin{array}{l}\vdash \mathtt{(typecase\ e_{sel}\ of} \\ \quad \mathtt{\ldots(\vec{X_i})\ \ (x_i{:}T_i)\ \ e_i\ldots} \\ \quad \mathtt{else\ e_{else})} \\ \ \ \mathtt{end)} \qquad\qquad\quad \Rightarrow \mathtt{v}\end{array}}$$

$$\frac{\vdash \mathtt{e} \Rightarrow \mathtt{v} \quad (\mathtt{v} \text{ not of the form } \mathtt{(dynamic\ w{:}T)})}{\begin{array}{l}\vdash \mathtt{(typecase\ e_{sel}\ of} \\ \quad \mathtt{\ldots(\vec{X_i})\ \ (x_i{:}T_i)\ \ e_i\ldots} \\ \quad \mathtt{else\ e_{else})} \\ \ \ \mathtt{end)} \qquad\qquad \Rightarrow \mathtt{wrong}\end{array}}$$

4.4 Soundness

We have defined two sets of rules—one for evaluating expressions and one for deriving their types. At this point, it is reassuring to observe that the two systems "fit together" in the way we would expect. We can show that "evaluation preserves typing"—that if a well-typed expression `e` reduces to a canonical expression `v`, then `v` is assigned the same type as `e` by the typing rules. From this it is an easy corollary that no well-typed program can evaluate to `wrong`.

We begin with a lemma that connects the form of proofs using the typing rules (which use type environments) with that of proofs using the evaluation rules (which use substitution instead of binding environments). Since many of the type environments we are concerned with will be empty, we write "$\vdash \mathtt{e} : \mathtt{T}$" as an abbreviation for "$\emptyset \vdash \mathtt{e} : \mathtt{T}$."

Lemma 4.4.1 (Substitution preserves typing) *For all expressions* `e`*, canonical expressions* `v`*, closed types* `V` *and* `W`*, type environments TE, and variables* `z`*, if* $\vdash \mathtt{v} : \mathtt{V}$ *and* $\mathit{TE}[\mathtt{z} \leftarrow \mathtt{V}] \vdash \mathtt{e} : \mathtt{W}$*, then* $\mathit{TE} \vdash \mathtt{e}[\mathtt{z} \leftarrow \mathtt{v}] : \mathtt{W}$.

Proof: We argue by induction on the length of a derivation of $\mathit{TE}[\mathtt{z} \leftarrow \mathtt{V}] \vdash \mathtt{e} : \mathtt{W}$. There is one case for each of the typing rules; in each case, we must show how to construct a derivation of $\mathit{TE} \vdash \mathtt{e}[\mathtt{z} \leftarrow \mathtt{v}] : \mathtt{W}$ from the a derivation whose final step is an application of the rule in question. We give the proof for three representative cases:

- $\mathtt{e} = \mathtt{x}$

  If $\mathtt{x} = \mathtt{z}$, then $\mathtt{e}[\mathtt{z} \leftarrow \mathtt{v}] = \mathtt{v}$. By the typing rule for variables, $\mathit{TE}[\mathtt{z} \leftarrow \mathtt{V}] \vdash \mathtt{z} : \mathtt{V}$. Immediately, $\mathit{TE} \vdash \mathtt{e}[\mathtt{z} \leftarrow \mathtt{v}] : \mathtt{V}$.

  If $\mathtt{x} \neq \mathtt{z}$, then $\mathtt{e}[\mathtt{z} \leftarrow \mathtt{v}] = \mathtt{x}$ and $\mathit{TE} \vdash \mathtt{e}[\mathtt{z} \leftarrow \mathtt{V}] : \mathtt{W}$.

- $\mathtt{e} = \lambda\mathtt{x:T.e}_{\mathrm{body}}$

  If $\mathtt{x} = \mathtt{z}$, then $\mathtt{e}[\mathtt{z} \leftarrow \mathtt{v}] = \mathtt{e}$. Immediately, $\mathit{TE} \vdash \mathtt{e}[\mathtt{z} \leftarrow \mathtt{v}] : \mathtt{W}$.

  If $\mathtt{x} \neq \mathtt{z}$, then for the typing rule for λ-expressions to apply (giving $\mathit{TE}[\mathtt{z} \leftarrow \mathtt{V}] \vdash \mathtt{e} : \mathtt{T}{\rightarrow}\mathtt{U}$ for some $\mathtt{T}$ and $\mathtt{U}$), it must be the case that $\mathit{TE}[\mathtt{x} \leftarrow \mathtt{T},\ \mathtt{z} \leftarrow \mathtt{V}] \vdash \mathtt{e}_{\mathrm{body}} : \mathtt{U}$. By the induction hypothesis, $\mathit{TE}[\mathtt{x} \leftarrow \mathtt{T}] \vdash \mathtt{e}_{\mathrm{body}}[\mathtt{z} \leftarrow \mathtt{v}] : \mathtt{U}$. By the typing rule for λ again, $\mathit{TE} \vdash \lambda\mathtt{x:T.(e}_{\mathrm{body}}[\mathtt{z} \leftarrow \mathtt{v}]\mathtt{)} : \mathtt{T}{\rightarrow}\mathtt{U}$. By the definition of substitution, $\mathit{TE} \vdash \mathtt{e}[\mathtt{z} \leftarrow \mathtt{v}] : \mathtt{T}{\rightarrow}\mathtt{U}$.

- $\mathtt{e} = \mathtt{e}_{\mathrm{fun}}\mathtt{(e}_{\mathrm{arg}}\mathtt{)}$

  For the typing rule for application to apply (giving $\mathit{TE}[\mathtt{z} \leftarrow \mathtt{V}] \vdash \mathtt{e} : \mathtt{W}$), it must be the case that $\mathit{TE}[\mathtt{s} \leftarrow \mathtt{v}] \vdash \mathtt{e}_{\mathrm{fun}} : \mathtt{T}{\rightarrow}\mathtt{W}$ and $\mathit{TE}[\mathtt{z} \leftarrow \mathtt{V}] \vdash \mathtt{e}_{\mathrm{arg}} : \mathtt{T}$ for some $\mathtt{T}$. By the induction hypothesis, $\mathit{TE} \vdash \mathtt{e}_{\mathrm{fun}}[\mathtt{z} \leftarrow \mathtt{v}] : \mathtt{T}{\rightarrow}\mathtt{W}$ and $\mathit{TE} \vdash \mathtt{e}_{\mathrm{arg}}[\mathtt{z} \leftarrow \mathtt{v}] : \mathtt{T}$. Now by the typing rule for application, $\mathit{TE} \vdash \mathtt{(e}_{\mathrm{fun}}[\mathtt{z} \leftarrow \mathtt{v}]\mathtt{)(e}_{\mathrm{arg}}[\mathtt{z} \leftarrow \mathtt{v}]\mathtt{)} : \mathtt{W}$. By the definition of substitution, $\mathit{TE} \vdash \mathtt{e}[\mathtt{z} \leftarrow \mathtt{v}] : \mathtt{W}$.

End of Proof.

Now we are ready for the soundness theorem itself.

Theorem 4.4.2 (Soundness) *For all expressions* $\mathtt{e}$, *canonical expressions* $\mathtt{v}$, *and types* $\mathtt{W}$, *if* $\vdash \mathtt{e} \Rightarrow \mathtt{v}$ *and* $\vdash \mathtt{e} : \mathtt{W}$, *then* $\vdash \mathtt{v} : \mathtt{W}$.

Proof: By induction on the length of the derivation $\vdash \mathtt{e} \Rightarrow \mathtt{v}$. There is one case for each possible syntactic form of $\mathtt{e}$. We show only a few representative cases:

- $\mathtt{e} = \lambda\mathtt{x:T.e}_{\mathrm{body}}$

  Immediate, since $\mathtt{v} = \mathtt{e}$.

- $\mathtt{e} = \mathtt{e}_{\mathrm{fun}}\mathtt{(e}_{\mathrm{arg}}\mathtt{)}$

  The typechecking rule for application must be the last step in the derivation of $\vdash \mathtt{e} : \mathtt{W}$, so $\vdash \mathtt{e}_{\mathrm{arg}} : \mathtt{T}$ and $\vdash \mathtt{e}_{\mathrm{fun}} : \mathtt{T}{\rightarrow}\mathtt{W}$ for some $\mathtt{T}$.

  If the last step in the derivation of $\vdash \mathtt{e} \Rightarrow \mathtt{v}$ is the second evaluation rule for application, then $\vdash \mathtt{e}_{\mathrm{fun}} \Rightarrow \mathtt{u}$ for some $\mathtt{u}$ not of the form $\lambda\mathtt{x:T.e}_{\mathrm{body}}$. But among canonical expressions, only those of this form are assigned a functional type by the typing rules, so our assumption contradicts the induction hypothesis.

  Similarly, if the last step in the derivation of $\vdash \mathtt{e} \Rightarrow \mathtt{v}$ is the third evaluation rule for application, then $\vdash \mathtt{e}_{\mathrm{arg}} \Rightarrow \mathtt{wrong}$. But $\mathtt{wrong}$ is not assigned any type whatsoever by the typing rules, again contradicting the induction hypothesis.

  So we may assume that the main evaluation rule for application is the last step in the derivation of $\vdash \mathtt{e} \Rightarrow \mathtt{v}$, from which it follows that $\vdash \mathtt{e}_{\mathrm{fun}} \Rightarrow \lambda\mathtt{x:T.e}_{\mathrm{body}}$, $\vdash \mathtt{e}_{\mathrm{arg}} \Rightarrow \mathtt{w}$ ($\mathtt{w} \neq \mathtt{wrong}$), and $\vdash \mathtt{e}_{\mathrm{body}}[\mathtt{x} \leftarrow \mathtt{w}] \Rightarrow \mathtt{v}$. By the induction hypothesis, $\vdash \mathtt{w} : \mathtt{T}$ and $\vdash \lambda\mathtt{x:T.e}_{\mathrm{body}} : \mathtt{T}{\rightarrow}\mathtt{W}$. Since the last step in the latter derivation must be the typing rule for λ-expressions, $[\mathtt{x} \leftarrow \mathtt{T}] \vdash \mathtt{e}_{\mathrm{body}} : \mathtt{W}$. By Lemma 4.4.1, $\vdash \mathtt{e}_{\mathrm{body}}[\mathtt{x} \leftarrow \mathtt{w}] : \mathtt{W}$. Finally, by the induction hypothesis again, $\vdash \mathtt{v} : \mathtt{W}$.

- $\mathtt{e} = \mathtt{dynamic}\ \mathtt{e}_{\mathrm{body}}: \ \mathtt{T}$

  If $\vdash \mathtt{e}_{\mathrm{body}} \Rightarrow \mathtt{wrong}$, then by the induction hypothesis and the typing rule for `dynamic`, $\vdash \mathtt{wrong} : \mathtt{T}$. This cannot be the case.

  So assume that $\vdash \mathtt{e}_{\mathrm{body}} \Rightarrow \mathtt{w}$ ($\mathtt{w} \neq \mathtt{wrong}$), so that the main evaluation rule for `dynamic` is the last step in the derivation of $\vdash \mathtt{e} \Rightarrow \mathtt{v}$. The typechecking rule for `dynamic` must be the last step in the derivation of $\vdash \mathtt{e} : \mathtt{W}$ (here $\mathtt{W} = \mathtt{Dynamic}$), so $\vdash \mathtt{e}_{\mathrm{body}} : \mathtt{T}$. By the induction hypothesis, $\vdash \mathtt{w} : \mathtt{T}$. By the typing rule for `dynamic` again, $\vdash \mathtt{v} : \mathtt{W}$.

- $\mathtt{e} = \mathtt{typecase}\ \mathtt{e}_{\mathrm{sel}}\ \mathtt{of}$

  $$\begin{array}{l} \quad\quad \mathtt{...} \\ \quad\quad (\vec{\mathtt{X}_{\mathrm{i}}})\ (\mathtt{x}_{\mathrm{i}}\mathtt{:T}_{\mathrm{i}})\ \mathtt{e}_{\mathrm{i}} \\ \quad\quad \mathtt{...} \\ \quad\quad \mathtt{else}\ \mathtt{e}_{\mathrm{else}} \\ \mathtt{end} \end{array}$$

  Assume that $\vdash \mathtt{e}_{\mathrm{sel}} \Rightarrow \mathtt{dynamic}\ \mathtt{w{:}U}$, that for some k, $\mathit{match}(\mathtt{U}, \mathtt{T}_{\mathrm{k}}) = \sigma$ while $\mathit{match}(\mathtt{U}, \mathtt{T}_{\mathrm{j}})$ fails for all $j < k$, and that $\mathtt{e}_{\mathrm{k}}\sigma[\mathtt{x}_{\mathrm{k}} \leftarrow \mathtt{w}] \Rightarrow \mathtt{v}$, so that the main evaluation rule for `typecase` is the last step in the derivation of $\vdash \mathtt{e} \Rightarrow \mathtt{v}$. (The argument for the second `typecase` rule is straighforward; the `wrong` case proceeds as in the previous two arguments.)

  By the typechecking rule for `typecase`, $\vdash \mathtt{e}_{\mathrm{sel}} : \mathtt{Dynamic}$. By the induction hypothesis, $\vdash \mathtt{w} : \mathtt{U}$. By the typechecking rule again, $[\mathtt{x}_{\mathrm{k}} \leftarrow \mathtt{T}_{\mathrm{k}}\sigma] \vdash \mathtt{e}_{\mathrm{k}}\sigma : \mathtt{W}$. By the definition of *match*, this can be rewritten as $[\mathtt{x}_{\mathrm{k}} \leftarrow \mathtt{U}] \vdash \mathtt{e}_{\mathrm{k}}\sigma : \mathtt{W}$. By Lemma 4.4.1, $\vdash \mathtt{e}_{\mathrm{k}}\sigma[\mathtt{x}_{\mathrm{k}} \leftarrow \mathtt{w}] : \mathtt{W}$. Now by the induction hypothesis, $\vdash \mathtt{v} : \mathtt{W}$.

End of Proof.

Since `wrong` is not assigned any type by the typing rules, the following is immediate:

Corollary 4.4.3 *For all expressions* $\mathtt{e}$*, canonical expressions* $\mathtt{v}$*, and types* $\mathtt{T}$*, if* $\vdash \mathtt{e} \Rightarrow \mathtt{v}$ *and* $\vdash \mathtt{e} : \mathtt{T}$ *then* $\mathtt{v} \neq \mathtt{wrong}$.

5 Denotational Semantics

Another way of showing that our rules are sound is to define a semantics for the language and show that no well-typed expression *denotes* `wrong`. In general terms, this involves constructing a domain **V** and defining a "meaning function" that assigns a value $[\![\mathtt{e}]\!]_\rho$ in **V** to each expression `e` in each environment ρ. The domain **V** should contain an element `wrong` such that $[\![\mathtt{wrong}]\!]_\rho = \mathtt{wrong}$ for all ρ.

Two properties are highly desirable:

- If `e` is a well-typed expression then $[\![\mathtt{e}]\!]_\rho \neq \mathtt{wrong}$ for well-behaved ρ.
- If $\vdash \mathtt{e} \Rightarrow \mathtt{v}$ then $[\![\mathtt{e}]\!]{=}[\![\mathtt{v}]\!]$ (that is, evaluation is sound).

To prove the former one, it suffices to map every typecode `T` to a subset $[\![\mathtt{T}]\!]$ of $\mathbf{V}$ not containing `wrong`, and prove:

- If $\vdash \mathtt{e} : \mathtt{T}$ then $[\![\mathtt{e}]\!]_\rho \in [\![\mathtt{T}]\!]$ for all ρ (that is, typechecking is sound).

In this section we carry out this program in an untyped model and suggest an approach with a typed model.

5.1 Untyped Semantics

In this subsection we give meaning to expressions as elements of an untyped universe $\mathbf{V}$ and to typecodes as subsets of $\mathbf{V}$. It would appear at first that the meaning of `Dynamic` can simply be defined as the set of all pairs $\langle v, \mathtt{T}\rangle$, such that $v \in [\![\mathtt{T}]\!]$. But `T` here ranges over all types, including `Dynamic` itself, so this definition as it stands is circular. We must build up the denotations of type expressions more carefully.

We therefore turn to the ideal model of types, following MacQueen, Plotkin, and Sethi [22]. (We refer the reader to this paper for the technical background of our construction.) Typecodes denote ideals—nonempty subsets of $\mathbf{V}$ closed under approximations and limits. We denote by **Idl** the set of all ideals in $\mathbf{V}$.

The ideal model has several features worth appreciating. First, to some extent the ideal model captures the intuition that types are sets of structurally similar values. Second, the ideal model accounts for diverse language constructs, including certain kinds of polymorphism. Finally, a large family of recursive type equations are guaranteed to have unique solutions. We exploit this feature to define the meaning of `Dynamic` with a recursive type equation.

We choose a universe $\mathbf{V}$ that satisfies the isomorphism equation

$$\mathbf{V} \cong \mathbf{N} + (\mathbf{V} \to \mathbf{V}) + (\mathbf{V} \times \mathit{TypeCode}) + \mathbf{W},$$

where $\mathbf{N}$ is the flat domain of natural numbers and $\mathbf{W}$ is the type error domain $\{w\}_\perp$. The usual continuous function space operation is represented as $\to$; the product-space $E \times A$ of a cpo E and a set A is defined as $\{\langle e, a\rangle \mid e \in E,\ e \neq \perp, \text{ and } a \in A\} \cup \{\perp_E\}$, with the evident ordering.

$\mathbf{V}$ can be obtained as the limit of a sequence of approximations $\mathbf{V}_0$, $\mathbf{V}_1$, ..., where

$$\begin{array}{lcl} \mathbf{V}_0 & = & \{\perp\} \\ \mathbf{V}_{i+1} & = & \mathbf{N} + (\mathbf{V}_i \to \mathbf{V}_i) + (\mathbf{V}_i \times \mathit{TypeCode}) + \mathbf{W}. \end{array}$$

We omit the details of the construction, which are standard [3, 22].

At this point, we have a universe suitable for assigning a meaning to expressions in our programming language. Figure 3 gives a full definition of the denotation function $[\![\ ]\!]$, using the following notation:

- "d in $\mathbf{V}$," where d belongs to a summand $\mathbf{S}$ of $\mathbf{V}$, is the injection of d into $\mathbf{V}$;
- `wrong` is an abbreviation for "w in $\mathbf{V}$";

$$\llbracket\ \rrbracket : \mathit{Exp}\to(\mathit{Var}\to\mathbf{V})\to\mathbf{V}$$

$$
\begin{aligned}
\llbracket \mathtt{x} \rrbracket_\rho &= \rho(\mathtt{x})\\
\llbracket \mathtt{wrong} \rrbracket_\rho &= \mathtt{wrong}\\
\llbracket \lambda\mathtt{x:T.e}_{\mathrm{body}} \rrbracket_\rho &= (\lambda v.\ \text{if } v = \mathtt{wrong} \text{ then } \mathtt{wrong} \text{ else } \llbracket \mathtt{e}_{\mathrm{body}} \rrbracket_{\rho\{x\,\leftarrow\,v\}}) \text{ in } \mathbf{V}\\
\llbracket \mathtt{e}_{\mathrm{fun}}\mathtt{(e}_{\mathrm{arg}}\mathtt{)} \rrbracket_\rho &= \text{if } \llbracket \mathtt{e}_{\mathrm{fun}} \rrbracket_\rho \not\sqsubseteq (\mathbf{V}\to\mathbf{V}) \text{ then } \mathtt{wrong} \text{ else } (\llbracket \mathtt{e}_{\mathrm{fun}} \rrbracket_\rho|_{\ \mathrm{V}\to\ \mathrm{V}})(\llbracket \mathtt{e}_{\mathrm{arg}} \rrbracket_\rho)\\
\\
\llbracket \mathtt{0} \rrbracket_\rho &= 0 \text{ in } \mathbf{V}\\
\llbracket \mathtt{succ\ e}_{\mathrm{nat}} \rrbracket_\rho &= \text{if } \llbracket \mathtt{e}_{\mathrm{nat}} \rrbracket_\rho \not\sqsubseteq \mathbf{N} \text{ then } \mathtt{wrong} \text{ else } (\llbracket \mathtt{e}_{\mathrm{nat}} \rrbracket_\rho|_{\ \mathrm{N}} + 1) \text{ in } \mathbf{V}\\
\llbracket \mathtt{test\ e}_{\mathrm{nat}}\ \mathtt{0:e}_{\mathrm{zero}}\ \mathtt{succ(x):e}_{\mathrm{succ}} \rrbracket_\rho &\\
&= \text{if } \llbracket \mathtt{e}_{\mathrm{nat}} \rrbracket_\rho \not\sqsubseteq \mathbf{N} \text{ then } \mathtt{wrong}\\
&\ \ \text{else if } \llbracket \mathtt{e}_{\mathrm{nat}} \rrbracket_\rho = 0 \text{ in } \mathbf{V} \text{ then } \llbracket \mathtt{e}_{\mathrm{zero}} \rrbracket_\rho\\
&\ \ \text{else } \llbracket \mathtt{e}_{\mathrm{succ}} \rrbracket_{\rho\{x\,\leftarrow\,((\llbracket \mathtt{e}_{\mathrm{nat}} \rrbracket_\rho|_{\ \mathrm{N}} - 1) \text{ in } \mathrm{V})\}}\\
\\
\llbracket \mathtt{dynamic\ e}_{\mathrm{body}}\mathtt{:T} \rrbracket_\rho &= \text{if } \llbracket \mathtt{e}_{\mathrm{body}} \rrbracket_\rho = \mathtt{wrong} \text{ then } \mathtt{wrong} \text{ else } (\langle \llbracket \mathtt{e}_{\mathrm{body}} \rrbracket_\rho, \mathtt{T}\rangle \text{ in } \mathbf{V})\\
\llbracket \mathtt{typecase\ e}_{\mathrm{sel}}\ \mathtt{of\ \ldots(}\vec{\mathtt{X}_{\mathrm{i}}}\mathtt{)(x}_{\mathrm{i}}\mathtt{:T}_{\mathrm{i}}\mathtt{)e}_{\mathrm{i}}\mathtt{\ldots else\ e}_{\mathrm{else}} \rrbracket_\rho &\\
&= \text{if } \llbracket \mathtt{e}_{\mathrm{sel}} \rrbracket_\rho \not\sqsubseteq (\mathbf{V}\times \mathit{TypeCode}) \text{ then } \mathtt{wrong}\\
&\ \ \text{else let } \langle d, \mathtt{U}\rangle = \llbracket \mathtt{e}_{\mathrm{sel}} \rrbracket_\rho|_{\ \mathrm{V}\times \mathit{TypeCode}} \text{ in}\\
&\ \ \quad \text{if } \ldots\\
&\ \ \quad \text{else if } \mathit{match}(\mathtt{U}, \mathtt{T}_{\mathrm{i}}) \text{ succeeds}\\
&\ \ \qquad \text{then let } \sigma = \mathit{match}(\mathtt{U}, \mathtt{T}_{\mathrm{i}}) \text{ in } \llbracket \mathtt{e}_{\mathrm{i}}\sigma \rrbracket_{\rho\{x_{\mathrm{i}}\leftarrow d\}}\\
&\ \ \quad \text{else if } \ldots\\
&\ \ \quad \text{else } \llbracket \mathtt{e}_{\mathrm{else}} \rrbracket_\rho
\end{aligned}
$$

Figure 3: The Meaning Function for Expressions

- $v|_{\mathbf{S}}$ yields: if $v = (d \text{ in } \mathbf{V})$ for some $d \in \mathbf{S}$ then d, otherwise $\bot$;
- $v \sqsubseteq \mathbf{S}$ yields $\bot$ if $v = \bot$, true if $v = (d \text{ in } \mathbf{V})$ for some $d \in \mathbf{S}$, and false otherwise;
- $=$ yields $\bot$ whenever either argument does.

Note that the definition of $=$ guarantees that $\llbracket \mathtt{(}\lambda\mathtt{x:T.e}_{\mathrm{body}}\mathtt{)(e}_{\mathrm{arg}}\mathtt{)} \rrbracket_\rho = \bot$ whenever $\llbracket \mathtt{e}_{\mathrm{arg}} \rrbracket_\rho = \bot$.

The denotation function "commutes" with substitutions and evaluation is sound with respect to the denotation function:

Lemma 5.1.1 *Let* $\mathtt{e}$ *be an expression,* σ *a substitution, and* ρ *and* ρ' *two environments. Assume that* ρ *maps each variable symbol* $\mathtt{x}$ *for which* σ *is defined to* $\llbracket \mathtt{x}\sigma \rrbracket_{\rho'}$*, and that it coincides with* ρ' *elsewhere. Then* $\llbracket \mathtt{e}\sigma \rrbracket_{\rho'} = \llbracket \mathtt{e} \rrbracket_\rho$.

Proof: The proof is a tedious inductive argument, and we omit it. *End of Proof.*

Theorem 5.1.2 *For all expressions* $\mathtt{e}$ *and* $\mathtt{v}$, *if* $\vdash \mathtt{e} \Rightarrow \mathtt{v}$ *then* $\llbracket \mathtt{e} \rrbracket = \llbracket \mathtt{v} \rrbracket$.

Proof: We argue by induction on the derivation of $\vdash \mathtt{e} \Rightarrow \mathtt{v}$. There is one case for each evaluation rule. We give only a few typical ones.

- For function applications: Assume that $\llbracket \mathtt{e}_{\mathrm{fun}} \rrbracket = \llbracket \lambda\mathtt{x}\mathtt{:T.}\mathtt{e}_{\mathrm{body}} \rrbracket$, $\llbracket \mathtt{e}_{\mathrm{arg}} \rrbracket = \llbracket \mathtt{w} \rrbracket$ with $\mathtt{w} \neq \mathtt{wrong}$, and $\llbracket \mathtt{e}_{\mathrm{body}}[\mathtt{x} \leftarrow \mathtt{w}] \rrbracket = \llbracket \mathtt{v} \rrbracket$, to prove that $\llbracket \mathtt{e}_{\mathrm{fun}}\mathtt{(}\mathtt{e}_{\mathrm{arg}}\mathtt{)} \rrbracket = \llbracket \mathtt{v} \rrbracket$. Note that $\llbracket \lambda\mathtt{x}\mathtt{:T.}\mathtt{e}_{\mathrm{body}} \rrbracket_{\rho}$ must be a function from $\mathbf{V}$ to $\mathbf{V}$ for all ρ, and $\mathtt{w}$ cannot denote $\mathtt{wrong}$ (since $\mathtt{w}$ is canonical). Therefore, we have $\llbracket \mathtt{e}_{\mathrm{fun}}\mathtt{(}\mathtt{e}_{\mathrm{arg}}\mathtt{)} \rrbracket_{\rho} = \llbracket \mathtt{e}_{\mathrm{body}} \rrbracket_{\rho\{x \leftarrow v\}}$ where $v = \llbracket \mathtt{e}_{\mathrm{arg}} \rrbracket_{\rho}$, for all ρ. Since $\llbracket \mathtt{e}_{\mathrm{arg}} \rrbracket = \llbracket \mathtt{w} \rrbracket$, Lemma 5.1.1 yields $\llbracket \mathtt{e}_{\mathrm{fun}}\mathtt{(}\mathtt{e}_{\mathrm{arg}}\mathtt{)} \rrbracket_{\rho} = \llbracket \mathtt{e}_{\mathrm{body}}[\mathtt{x} \leftarrow \mathtt{w}] \rrbracket_{\rho}$, and the hypothesis $\llbracket \mathtt{e}_{\mathrm{body}}[\mathtt{x} \leftarrow \mathtt{w}] \rrbracket = \llbracket \mathtt{v} \rrbracket$ immediately leads to the desired equation.

- For construction of dynamic values: Assume that $\llbracket \mathtt{e}_{\mathrm{body}} \rrbracket = \llbracket \mathtt{w} \rrbracket$ with $\mathtt{w} \neq \mathtt{wrong}$, to prove that $\llbracket \mathtt{dynamic}\ \mathtt{e}_{\mathrm{body}}\mathtt{:T} \rrbracket = \llbracket \mathtt{dynamic}\ \mathtt{w:T} \rrbracket$. As in the previous case, because $\mathtt{w}$ cannot denote $\mathtt{wrong}$, we have $\llbracket \mathtt{dynamic}\ \mathtt{e}_{\mathrm{body}}\mathtt{:T} \rrbracket_{\rho} = \langle \llbracket \mathtt{e}_{\mathrm{body}} \rrbracket_{\rho}, \mathtt{T} \rangle$ and $\llbracket \mathtt{dynamic}\ \mathtt{w:T} \rrbracket_{\rho} = \langle \llbracket \mathtt{w} \rrbracket_{\rho}, \mathtt{T} \rangle$. The desired equation follows at once from $\llbracket \mathtt{e}_{\mathrm{body}} \rrbracket = \llbracket \mathtt{w} \rrbracket$.

- For typecase operations: Assume that $\llbracket \mathtt{e}_{\mathrm{sel}} \rrbracket = \llbracket \mathtt{dynamic}\ \mathtt{w:T} \rrbracket$, $\mathit{match}(\mathtt{T}, \mathtt{T}_{\mathrm{j}})$ fails for all $j < k$, $\mathit{match}(\mathtt{T}, \mathtt{T}_{\mathrm{k}}) = \sigma$, and $\llbracket \mathtt{e}_{\mathrm{k}}\sigma\ [\mathtt{x}_{\mathrm{k}} \leftarrow \mathtt{w}] \rrbracket = \llbracket \mathtt{v} \rrbracket$, to prove that $\llbracket \mathtt{typecase}\ \mathtt{e}_{\mathrm{sel}}\ \mathtt{of}\ \ldots(\vec{\mathtt{X}}_{\mathrm{i}})\ (\mathtt{x}_{\mathrm{i}}\mathtt{:T}_{\mathrm{i}})\ \mathtt{e}_{\mathrm{i}}\ldots\mathtt{else}\ \mathtt{e}_{\mathrm{else}}\ \mathtt{end} \rrbracket = \llbracket \mathtt{v} \rrbracket$. As usual, $\mathtt{w}$ cannot denote $\mathtt{wrong}$, and hence we obtain the following chain of equalities, for arbitrary ρ: $\llbracket \mathtt{typecase}\ \mathtt{e}_{\mathrm{sel}}\ \mathtt{of}\ \ldots(\vec{\mathtt{X}}_{\mathrm{i}})\ (\mathtt{x}_{\mathrm{i}}\mathtt{:T}_{\mathrm{i}})\ \mathtt{e}_{\mathrm{i}}\ldots\mathtt{else}\ \mathtt{e}_{\mathrm{else}}\ \mathtt{end} \rrbracket_{\rho}$ equals $\llbracket \mathtt{e}_{\mathrm{k}}\sigma \rrbracket_{\rho\{x_{\mathrm{k}} \leftarrow d\}}$, where d is $\llbracket \mathtt{w} \rrbracket_{\rho}$ (by the hypotheses and the definition of $\llbracket\ \rrbracket$), equals $\llbracket \mathtt{e}_{\mathrm{k}}\sigma[\mathtt{x}_{\mathrm{k}} \leftarrow \mathtt{w}] \rrbracket_{\rho}$ (by Lemma 5.1.1), equals $\llbracket \mathtt{v} \rrbracket_{\rho}$ (by the hypotheses). The case where the `else` branch of a `typecase` is chosen is similar but simpler.

End of Proof.

Although we now have a meaning $\llbracket \mathtt{e} \rrbracket$ for each program $\mathtt{e}$, we do not yet have a meaning $\llbracket \mathtt{T} \rrbracket$ for each typecode $\mathtt{T}$. Therefore, in particular, we cannot prove yet that typechecking is sound. The main difficulty, of course, is to decide on the meaning of `Dynamic`.

We define the type of dynamic values with a recursive equation. Some auxiliary operations are needed to write this equation.

Definition 5.1.3 *If* $I \subseteq \mathbf{V}$ *is a set of values and* $\mathtt{T}$ *is a typecode, then*

$$I_{\mathtt{T}} = \{c \mid \langle c, \mathtt{T} \rangle \in I\}.$$

(Often, and in these definitions in particular, we omit certain injections from summands into $\mathbf{V}$ and the corresponding projections from $\mathbf{V}$ to its summands, which can be recovered from context.)

Definition 5.1.4 *If* $I \subseteq \mathbf{V}$ *and* $J \subseteq \mathbf{V}$ *are two sets of values, then*

$$I \twoheadrightarrow J = \{\langle c, \mathtt{T}{\to}\mathtt{U} \rangle \mid c(I_{\mathtt{T}}) \subseteq J_{\mathtt{U}},\ \mathit{where}\ \mathtt{T}, \mathtt{U} \in \mathit{TypeCode}\}.$$

Note that if I and J are ideals then so is $I \twoheadrightarrow J$.

Using these definitions, we can write an equation for the type of dynamic values:

$$\begin{array}{rll}\mathbf{D} & = & \mathbf{N} \times \{\mathtt{Nat}\} \\ & \cup & \mathbf{D} \twoheadrightarrow \mathbf{D} \\ & \cup & \mathbf{D} \times \{\mathtt{Dynamic}\}\end{array}$$

Here the variable $\mathbf{D}$ ranges over **Idl**, the set of all ideals in $\mathbf{V}$.

The equation follows from our informal definition of the type of dynamic values as the set of pairs $\langle \mathtt{v}, \mathtt{T}\rangle$ where $[\![\mathtt{v}]\!] \in [\![\mathtt{T}]\!]$. Intuitively, the equation states that a dynamic value can be one of three things. First, a dynamic value with tag `Nat` must contain a natural number. Second, if $\langle c, \mathtt{T}\to\mathtt{U}\rangle$ is a dynamic value then $c(v) \in [\![\mathtt{U}]\!]$ for all $v \in [\![\mathtt{T}]\!]$, and hence $\langle c(v), \mathtt{U}\rangle$ is a dynamic value whenever $\langle v, \mathtt{T}\rangle$ is. Third, a dynamic value with tag `Dynamic` must contain a dynamic value.

How is one to guarantee that this equation actually defines the meaning of `Dynamic`? MacQueen, Plotkin, and Sethi invoke the Banach Fixed Point Theorem to show that equations of the form $\mathbf{D} = F(\mathbf{D})$ over **Idl** have unique solutions, provided F is *contractive* in the following sense.

Informally, the rank $r(a)$ of an element a of $\mathbf{V}$ is the least i such that a "appears" in $\mathbf{V}_i$ during the construction of $\mathbf{V}$ as a limit. A witness for two ideals I and J is an element that belongs to one but not to the other; their distance $d(I, J)$ is 2^{-r}, where r is the minimum rank of a witness for the ideals. The function G is contractive if there exists a real number $t < 1$ such that for all $X_1, \ldots, X_n, X'_1, \ldots, X'_n$, we have

$$d(G(X_1, \ldots, X_n), G(X'_1, \ldots, X'_n)) \;\leq\; t \cdot max\{d(X_i, X'_i) \mid 1 \leq i \leq n\}.$$

Typically, one guarantees that an operation is contractive by expressing it in terms of basic operations such as × and →, and then inspecting the structure of this expression. In our case, we have a new basic operation, $\twoheadrightarrow$; in addition, × is slightly nonstandard. We need to prove that these two operations are contractive.

Theorem 5.1.5 *The operation × is contractive (when its second argument is fixed). The operation $\twoheadrightarrow$ is contractive.*

Proof: The arguments are based on the corresponding ones for Theorem 7 of [22]. In fact, the proof for × is a trivial variant of the corresponding one. We give only the proof for $\twoheadrightarrow$.

Let c be a witness of minimum rank for $I \twoheadrightarrow J$ and $I' \twoheadrightarrow J'$, being, say, only in the former ideal. Then $c \neq \bot$ (otherwise it would not be a witness), so $c = \langle f, \mathtt{T}\to\mathtt{U}\rangle$ for some f, `T`, and `U`. By the analogue of Proposition 4 of [22], $f = \bigsqcup(a_i \Rightarrow b_i)$ for some $a_i, b_i \in \mathbf{V}$, with $r(f) > \max(r(a_i), r(b_i))$ (here $a_i \Rightarrow b_i$ denotes the step function which returns b_i for arguments larger than a_i and $\bot$ otherwise). Since $c \notin I' \twoheadrightarrow J'$, f is not in $I'_{\mathtt{T}} \to J'_{\mathtt{U}}$. Hence there must be an $x \in I'_{\mathtt{T}}$ such that $f(x) \notin J'_{\mathtt{U}}$. Let $a = \bigsqcup\{a_i \mid a_i \sqsubseteq x\}$ and $b = \bigsqcup\{b_i \mid a_i \sqsubseteq x\} = f(x)$. Then $a \in I'_{\mathtt{T}}$ (since $a \sqsubseteq x$) but $b \notin J'_{\mathtt{U}}$. Moreover, by the analogue of Proposition 4 of [22], $r(a) \leq max\{r(a_i) \mid a_i \sqsubseteq x\} < r(f)$ and $r(b) < r(f)$. Similarly, $r(a) + 1 < r(c)$ and $r(b) + 1 < r(c)$.

There are two cases. If $a \notin I_{\mathtt{T}}$ then $\langle a, \mathtt{T}\rangle$ is a witness for I and I' of rank less than $r(c)$. (For all v, $r(\langle v, \mathtt{T}\rangle) \leq r(v) + 1$.) Otherwise, $a \in I_{\mathtt{T}}$ and so $b = f(a) \in J_{\mathtt{U}}$ since

$f \in I_{\mathrm{T}} \to J_{\mathrm{U}}$. Thus $\langle b, \mathtt{U} \rangle$ is a witness for J and J' of rank less than $r(c)$. In either case, we have $c(I \twoheadrightarrow J,\ I' \twoheadrightarrow J') = r(c) > min(c(I, I'),\ c(J, J'))$. *End of Proof.*

Immediately, the general result about the existence of fixed points yields the desired theorem.

Theorem 5.1.6 *The equation*

$$\begin{array}{rcl} \mathbf{D} & = & \mathbf{N} \times \{\mathtt{Nat}\} \\ & \cup & \mathbf{D} \twoheadrightarrow \mathbf{D} \\ & \cup & \mathbf{D} \times \{\mathtt{Dynamic}\} \end{array}$$

has a unique solution in **Idl**.

Let us call this solution **Dynamic**.

$$\boxed{\begin{array}{l} [\![\ \ ]\!] : \mathit{TypeCode} \to \mathbf{Idl} \\ \\ \begin{array}{lcl} [\![\mathtt{Nat}]\!] & = & \mathbf{N} \\ [\![\mathtt{Dynamic}]\!] & = & \mathbf{Dynamic} \\ [\![\mathtt{T} \to \mathtt{U}]\!] & = & \{c \mid c([\![\mathtt{T}]\!]) \subseteq [\![\mathtt{U}]\!]\} \end{array} \end{array}}$$

Figure 4: The Meaning Function for Typecodes

Finally, we are in a position to associate an ideal $[\![\mathtt{T}]\!]$ with each typecode `T` (see figure 4). The semantics fits our original intuition of what dynamic values are, as the following lemma shows.

Lemma 5.1.7 *For all values v and typecodes* `T`, $\langle v, \mathtt{T} \rangle \in$ **Dynamic** *if and only if* $v \in [\![\mathtt{T}]\!]$.

Proof: The proof is by induction on the structure of `T`.

For $\mathtt{T} = \mathtt{Nat}$, we need to check that $\langle v, \mathtt{Nat} \rangle \in$ **Dynamic** if and only if $v \in [\![\mathtt{Nat}]\!]$. This follows immediately from the equation, since all and only natural numbers are tagged with `Nat`.

Similarly, for $\mathtt{T} = \mathtt{Dynamic}$, we need to check that $\langle v, \mathtt{Dynamic} \rangle \in$ **Dynamic** if and only if $v \in [\![\mathtt{Dynamic}]\!]$. This follows immediately from the equation, since all and only dynamic values are tagged with `Dynamic`.

Finally, for $\mathtt{T} = \mathtt{U} \to \mathtt{V}$, we need to check that $\langle v, \mathtt{U} \to \mathtt{V} \rangle \in$ **Dynamic** if and only if $v \in [\![\mathtt{U} \to \mathtt{V}]\!]$. By induction hypothesis, we have $\mathbf{Dynamic}_{\mathtt{U}} = [\![\mathtt{U}]\!]$ and $\mathbf{Dynamic}_{\mathtt{V}} = [\![\mathtt{V}]\!]$. We derive the following chain of equivalences: $\langle v, \mathtt{U} \to \mathtt{V} \rangle \in$ **Dynamic** if and only if $v(\mathbf{Dynamic}_{\mathtt{U}}) \subseteq \mathbf{Dynamic}_{\mathtt{V}}$ (according to the equation), if and only if $v([\![\mathtt{U}]\!]) \subseteq [\![\mathtt{V}]\!]$ (by induction hypothesis), if and only if $v \in [\![\mathtt{U} \to \mathtt{V}]\!]$ (according to the definition of $[\![\ \ ]\!]$). *End of Proof.*

We can also prove the soundness of typechecking:

Definition 5.1.8 *The environment ρ is consistent with the type environment TE on the expression* $\mathtt{e}$ *if $TE(\mathtt{x})$ is defined and $\rho(\mathtt{x}) \in [\![TE(\mathtt{x})]\!]$ for all $\mathtt{x} \in FV(\mathtt{e})$.*

Theorem 5.1.9 *For all type environments TE, expressions* $\mathtt{e}$*, environments ρ consistent with TE on* $\mathtt{e}$*, and typecodes* $\mathtt{T}$*, if $TE \vdash \mathtt{e} : \mathtt{T}$ then $[\![\mathtt{e}]\!]_\rho \in [\![\mathtt{T}]\!]$.*

Proof: We argue by induction on the derivation of $TE \vdash \mathtt{e} : \mathtt{T}$. There is one case for each typing rule. We give only a few typical ones.

- For abstractions: Assume that $[\![\mathtt{e}]\!]_\rho \in \mathtt{T}$ for all TE and all ρ consistent with $TE[\mathtt{x} \leftarrow \mathtt{U}]$ on $\mathtt{e}$, to prove that $[\![\lambda\mathtt{x}:\mathtt{U}.\mathtt{e}]\!]_\rho \in (\mathtt{U}\to\mathtt{T})$ for all TE and all ρ consistent with TE on $\lambda\mathtt{x}:\mathtt{U}.\mathtt{e}$. Consider some $v \in [\![\mathtt{U}]\!]$. According to the definition of $[\![\ ]\!]$, we need to show that $([\![\lambda\mathtt{x}:\mathtt{U}.\mathtt{e}]\!]_\rho)v \in [\![\mathtt{T}]\!]$. We may assume that $v \neq \perp$ (the $\perp$ case is trivial), and $v \neq \mathtt{wrong}$ ($[\![\mathtt{U}]\!]$ cannot contain $\mathtt{wrong}$). Thus, we have $[\![\lambda\mathtt{x}:\mathtt{U}.\mathtt{e}]\!]_\rho v = [\![\mathtt{e}]\!]_{\rho\{\mathtt{x}\,\leftarrow\,v\}}$. The hypothesis immediately yields that this value is a member of $[\![\mathtt{T}]\!]$.

- For function applications: Assume that $[\![\mathtt{e}_{\mathrm{fun}}]\!]_\rho \in [\![\mathtt{U}\to\mathtt{T}]\!]$ and that $[\![\mathtt{e}_{\mathrm{arg}}]\!]_\rho \in [\![\mathtt{U}]\!]$, to prove that $[\![\mathtt{e}_{\mathrm{fun}}(\mathtt{e}_{\mathrm{arg}})]\!]_\rho \in [\![\mathtt{T}]\!]$. By the definition of function types, $[\![\mathtt{e}_{\mathrm{fun}}]\!]_\rho$ must be a function from $\mathbf{V}$ to $\mathbf{V}$, and $[\![\mathtt{e}_{\mathrm{arg}}]\!]_\rho$ cannot be $\mathtt{wrong}$, since $[\![\mathtt{U}]\!]$ cannot contain $\mathtt{wrong}$. In addition, we may assume that $[\![\mathtt{e}_{\mathrm{arg}}]\!]_\rho \neq \perp$ (the $\perp$ case is trivial). Immediately, $[\![\mathtt{e}_{\mathrm{fun}}(\mathtt{e}_{\mathrm{arg}})]\!]_\rho = ([\![\mathtt{e}_{\mathrm{fun}}]\!]_\rho)[\![\mathtt{e}_{\mathrm{arg}}]\!]_\rho$, and the definition of $[\![\mathtt{U}\to\mathtt{T}]\!]$ yields that this value must be a member of $[\![\mathtt{T}]\!]$.

- For construction of dynamic values: Assume that $[\![\mathtt{e}_{\mathrm{body}}]\!]_\rho \in [\![\mathtt{T}]\!]$, to prove that $[\![\mathtt{dynamic}\ \mathtt{e}_{\mathrm{body}}:\mathtt{T}]\!]_\rho \in [\![\mathtt{Dynamic}]\!]$. Since $[\![\mathtt{T}]\!]$ cannot contain $\mathtt{wrong}$, $[\![\mathtt{e}_{\mathrm{body}}]\!]_\rho \neq \mathtt{wrong}$, and hence $[\![\mathtt{dynamic}\ \mathtt{e}_{\mathrm{body}}:\mathtt{T}]\!]_\rho = \langle[\![\mathtt{e}_{\mathrm{body}}]\!]_\rho, \mathtt{T}\rangle$. The desired result then follows from Lemma 5.1.7.

- For typecase operations: Assume that $[\![\mathtt{e}_{\mathrm{sel}}]\!]_\rho \in \mathbf{Dynamic}$ for all TE and all ρ consistent with TE on $\mathtt{e}_{\mathrm{sel}}$; $[\![\mathtt{e}_{\mathrm{i}}\sigma]\!]_\rho \in [\![\mathtt{T}]\!]$ for all i, for all $\sigma \in \mathit{Subst}_{\vec{\mathtt{X}_{\mathrm{i}}}}$; and for all TE and all ρ consistent with $TE[\mathtt{x}_{\mathrm{i}} \leftarrow \mathtt{T}_{\mathrm{i}}\sigma]$ on $\mathtt{e}_{\mathrm{i}}\sigma$, and $[\![\mathtt{e}_{\mathrm{else}}]\!]_\rho \in [\![\mathtt{T}]\!]$ for all TE and all ρ consistent with TE on $\mathtt{e}_{\mathrm{else}}$. We prove that $[\![\mathtt{typecase}\ \mathtt{e}_{\mathrm{sel}}\ \mathtt{of}\ \ldots(\vec{\mathtt{X}_{\mathrm{i}}})\ (\mathtt{x}_{\mathrm{i}}:\mathtt{T}_{\mathrm{i}})\ \mathtt{e}_{\mathrm{i}}\ldots\mathtt{else}\ \mathtt{e}_{\mathrm{else}}\ \mathtt{end}]\!]_\rho \in [\![\mathtt{T}]\!]$ for all TE and all ρ consistent with TE on $(\mathtt{typecase}\ \mathtt{e}_{\mathrm{sel}}\ \mathtt{of}\ \ldots(\vec{\mathtt{X}_{\mathrm{i}}})(\mathtt{x}_{\mathrm{i}}:\mathtt{T}_{\mathrm{i}})\mathtt{e}_{\mathrm{i}}\ldots\mathtt{else}\ \mathtt{e}_{\mathrm{else}}\ \mathtt{end})$. Similarly to the other cases, $[\![\mathtt{e}_{\mathrm{sel}}]\!]_\rho$ must be the pair of a value and a typecode, and we may assume that it is not $\perp$. Hence, $[\![\mathtt{typecase}\ \mathtt{e}_{\mathrm{sel}}\ \mathtt{of}\ \ldots(\vec{\mathtt{X}_{\mathrm{i}}})\ (\mathtt{x}_{\mathrm{i}}:\mathtt{T}_{\mathrm{i}})\ \mathtt{e}_{\mathrm{i}}\ldots\mathtt{else}\ \mathtt{e}_{\mathrm{else}}\ \mathtt{end}]\!]_\rho$ is either $[\![\mathtt{e}_{\mathrm{i}}\sigma]\!]_{\rho\{\mathtt{x}_{\mathrm{i}}\leftarrow d\}}$ for some i and with d equal to the first component of the selector, or simply $[\![\mathtt{e}_{\mathrm{else}}]\!]_\rho$. In the former case, Lemma 5.1.7 guarantees that $d \in [\![\mathtt{T}_{\mathrm{i}}\sigma]\!]$, and hence the hypotheses guarantee that $[\![\mathtt{e}_{\mathrm{i}}\sigma]\!]_{\rho\{x_{\mathrm{i}}\leftarrow d\}} \in [\![\mathtt{T}]\!]$. In the latter case, the hypotheses guarantee that $[\![\mathtt{e}_{\mathrm{else}}]\!]_\rho \in [\![\mathtt{T}]\!]$. In either case, we derive $[\![\mathtt{typecase}\ \mathtt{e}_{\mathrm{sel}}\ \mathtt{of}\ \ldots(\vec{\mathtt{X}_{\mathrm{i}}})\ (\mathtt{x}_{\mathrm{i}}:\mathtt{T}_{\mathrm{i}})\ \mathtt{e}_{\mathrm{i}}\ldots\mathtt{else}\ \mathtt{e}_{\mathrm{else}}\ \mathtt{end}]\!]_\rho \in [\![\mathtt{T}]\!]$.

End of Proof.

It follows from Theorem 5.1.2, Theorem 5.1.9, and the fact that no $[\![\mathtt{T}]\!]$ can contain (w in $\mathbf{V}$) that no well-typed expression evaluates to $\mathtt{wrong}$. This gives us a new proof of Corollary 4.4.3.

5.2 Typed Semantics

The semantics $[\![\ ]\!]$ is, essentially, a semantics for the untyped lambda-calculus, as in its definition type information is ignored. This seems very appropriate for languages with implicit typing, where some or all of the type information is omitted in programs. But for an explicitly typed language it seems natural to look for a semantics that assigns elements of domains $\mathbf{V}_{\mathtt{T}}$ to expressions of type `T`. One idea to find these domains is to solve the infinite set of simultaneous equations

$$\begin{aligned} \mathbf{V}_{\mathtt{Nat}} &= \mathbf{N} \\ \mathbf{V}_{\mathtt{T}\rightarrow\mathtt{U}} &= \mathbf{V}_{\mathtt{T}}\rightarrow\mathbf{V}_{\mathtt{U}} \\ \mathbf{V}_{\mathtt{Dynamic}} &= \sum_{\mathtt{T}} \mathbf{V}_{\mathtt{T}} \end{aligned}$$

A similar use of sums appears in Mycroft's work [28].

6 Extensions

In this section we present some preliminary thoughts on extending the ideas in the rest of the paper to languages with implicit or explicit polymorphism, abstract data types, and more expressive type patterns.

6.1 Polymorphism

For most of the section, we assume an explicitly typed polymorphic lambda calculus along the lines of Reynolds' system [32]. The type abstraction operator is written as Λ. Type application is written with square brackets. The types of polymorphic functions begin with $\forall$. For example, $\forall\mathtt{T}.\mathtt{T}\rightarrow\mathtt{T}$ is the type of the polymorphic identity function, $\Lambda\mathtt{T}.\lambda\mathtt{x}{:}\mathtt{T}.\ \mathtt{x}$.

In the simplest case, the typechecking and operational semantics of `dynamic` and `typecase` carry over nearly unchanged from the language described in Section 4. We simply redefine *match* as follows:

> If there is a substitution σ such that `T` and $\mathtt{U}\sigma$ are identical up to renaming of bound type variables, then *match*(`T`, `U`) returns some such substitution. Otherwise, *match*(`T`, `U`) fails.

We can now write `typecase` expressions that match polymorphic type tags. For example, the following function checks that `f` is a polymorphic function taking elements of any type into `Nat`. It then instantiates `f` at `W`, the type tag of its second argument, and applies the result to the value part of the second argument.

```
λdf:Dynamic. λde:Dynamic.
  typecase df of
    (f: ∀Z. Z → Nat)
      typecase de of
        (W) (e: W)  f[W](e)
        else 0
```

```
        end
      else 0
    end
```

6.2 Abstract Data Types

In a similar vein, we can imagine extending the language of type tags to include existentially quantified variables. Following Mitchell and Plotkin [27], we can think of a `Dynamic` whose tag is an existential type as being a module with hidden implementation, or alternatively as an encapsulated element of an abstract data type. Our notation for existential types and labeled products follows that of Cardelli and Wegner [11]. For example,

```
λs: ∃Rep. {push:  Rep → Nat → Rep,
           pop:   Rep → (Nat × Rep),
           top:   Rep → Nat,
           empty: Rep}.
         open s as stk[Rep]
             in stk.top(stk.push stk.empty 5)
```

is a function that takes a stack package (a tuple containing a hidden representation type `Rep`, three functions, and a constant value), opens the package (making its components accessible in the body of the `open` expression), and performs the trivial computation of pushing the number `5` onto an empty stack and returning the top element of the resulting stack.

The following function takes a `Dynamic` containing a stack package (with hidden representation) and another `Dynamic` of the same type as the elements of the stack. It pushes its second argument onto the empty stack from the stack package, and returns the top of the resulting stack, appropriately repackaged as a dynamic value.

```
λds:Dynamic. λde:Dynamic.
    typecase ds of
      (X) (s: ∃Rep.
                {push:  Rep → X → Rep,
                 pop:   Rep → (X × Rep),
                 top:   Rep → X,
                 empty: Rep})
            typecase de of
              (e: X)  open s as stk[Rep]
                          in dynamic stk.top(stk.push stk.empty e) : X
              else e
            end
      else e
    end
```

In order to preserve the integrity of existentially quantified values in a language that also has `Dynamic`, it seems necessary to place some restrictions on the types that may appear in `dynamic` expressions to prevent their being used to expose the witness type of an existentially quantified value beyond the scope of an `open` (or `abstype`) block. In particular, the type tag in a `dynamic` constructor must not be allowed to mention the representation types of any currently open abstract data types, as in the following:

```
λds:Dynamic. λde:Dynamic.
   typecase ds of
     (X) (s: ∃Rep.
                {push:  Rep → X → Rep,
                 pop:   Rep → (X × Rep),
                 top:   Rep → X,
                 empty: Rep})
             open s as stk[Rep]
                in (* Wrong: *) dynamic stk.empty : Rep
     else de
   end
```

It would be wrong here to create a Dynamic whose type tag is the representation type of the stack (assuming such type is available at run-time), because this would violate the abstraction. It is also unclear how to generate a type tag that does not violate the abstraction. Hence we choose to forbid this situation.

6.3 Restrictions

In a language with both explicit polymorphism and `Dynamic`, it is possible to write programs where types must actually be passed to functions at run time:

```
ΛX. λx:X. dynamic x:X
```

The extra cost of actually performing type abstractions and applications at run time (rather than just checking them during compilation and then discarding them) should not be prohibitive. Still, we might also want to consider how the `dynamic` construct might be restricted so that types need not be passed around during execution. A suitable restriction is that an expression `dynamic e:T` is well-formed only if `T` is closed.

This restriction was proposed by Mycroft [28] in the context of an extension of ML, which uses implicit rather than explicit polymorphism. The appropriate analogue of "closed type expressions" in ML is "type expressions with only generic type variables"—expressions whose type variables are either instantiated to some known type or else totally undetermined (that is, not dependent on any type variable whose value is unknown at compile time).

In fact, in languages with implicit polymorphism, Mycroft's restriction on `dynamic` is *required*: there is no natural way to determine where the type applications should be performed at run time. Dynamics with non-generic variables can be used to break the ML type system. (The problem is analogous to that of "updateable refs" [37].)

6.4 Higher-order Pattern Variables

By enriching the language of type patterns, it is possible to express a much broader range of computations on `Dynamic`s, including some interesting ones involving polymorphic functions. Our motivating example here is a generalization of the dynamic application function from Section 3. The problem there is to take two dynamic values, make sure that the first is a function and the second an argument belonging to the function's domain, and apply the function. Here we want to allow the first argument to be a polymorphic function and narrow it to an appropriate monomorphic instance automatically, before applying it to the supplied parameter. We call this "polymorphic dynamic application."

To express this example, we need to extend the `typecase` construct with "functional" pattern variables. Whereas ordinary pattern variables range over type expressions, functional pattern variables (named `F`, `G`, etc., to distinguish them from ordinary pattern variables) range over functions from type expressions to type expressions.

Using functional pattern variables, polymorphic dynamic application can be expressed as follows:

```
λdf:Dynamic. λde:Dynamic.
  typecase df of
    (F,G) (f: ∀Z. (F Z) → (G Z))
      typecase de of
        (W) (e: (F W))
          dynamic f[W](e):(G W)
        else
          dynamic "Error":String
      end
    else
      dynamic "Error":String
  end
```

For instance, when we apply the function to the arguments

```
df = dynamic (ΛZ.λx:Z.x): (∀Z. Z→Z)
de = dynamic 3:Nat
```

the first branch of the outer typecase succeeds, binding `F` and `G` to the identity function on type expressions. The first branch of the inner typecase succeeds, binding `W` to `Nat` so that `(F W)` = `Nat` and `(G W)` = `Nat`. Now `f(F W)` reduces to `λx:(F W).x` and `f(G W)(e)` reduces to 3, which has type `(G W)` = `Nat` as claimed.

Another intriguing example is polymorphic dynamic composition:

```
λdf:Dynamic. λdg:Dynamic.
  typecase df of
    (F,G) (f: ∀W. (F W)→(G W))
      typecase dg of
        (H) (g: ∀V. (G V)→(H V))
          dynamic (ΛW. g[W] ∘ f[W])
                      : ∀V.(F V)→(G V)
        else ...
      end
    else ...
  end
```

This function checks that its two arguments are both polymorphic functions and that their composition is well-typed, returning the composition if so.

6.5 Open Issues

This preliminary treatment of polymorphism and higher-order pattern variables leaves a number of questions unanswered: What is the appropriate specification for the *match* operation? How difficult is it to compute? Is there a sensible notion of "most general substitution" when pattern variables can range over things like functions from type expressions to type expressions? Should pattern variables range over *all* functions from type expressions to type expressions, or only over some more restricted class of functions? What are the implications (for both operational and denotational semantics) of implicit vs. explicit polymorphism? We hope that our examples may stimulate the creativity of others in helping to answer these questions.

7 Implementation Issues

This section discusses some of the issues that arise in implementations of languages with dynamic values and a `typecase` construct: methods for efficient transfer of dynamic values to and from persistent storage, implementation of the *match* function, and representation of type tags for efficient matching.

7.1 Persistent Storage

One of the most important purposes of dynamic values is as a safe and uniform format for persistent data. This facility may be heavily exploited in large software environments, so it is important that it be implemented efficiently. Large data structures, possibly with circularities and shared substructures, need to be represented externally so that they can be quickly rebuilt in the heap of a running program. (The type tags present no special difficulties: they are ordinary run-time data structures.)

Fortunately, a large amount of energy has already been devoted to this problem, particularly in the Lisp community. Many Lisp systems support "fasl" files, which can be used to store arbitrary heap structures. (See [24] for a description of a typical fasl format. The idea goes back to 1974, at least.)

A mechanism for "pickling" heap structures in Cedar/Mesa was designed and implemented by Rovner and Maxwell, probably in 1982 or 1983. A variant of their algorithm, due to Lampson, is heavily used in the Modula-2+ programming environment at the DEC Systems Research Center. Another scheme was implemented as part of Tartan Labs' Interface Description Language [29]. This scheme was based on earlier work by Newcomer and Dill on the "Production Quality Compiler-Compiler" project at CMU.

7.2 Type Matching

Although the particular language constructs described in this paper have not been implemented, various schemes for dynamic typing in statically typed languages have existed for some time (see Section 2). Figure 5 gives a rough classification of several languages according to the amount of work involved in comparing types and the presence or absence of subtyping.

	Without subtyping	With subtyping
Name equivalence	Modula-2+, CLU, etc.	Simula-67
Rigid Structural Equivalence		Modula-3, Cedar
Structural Equivalence		Amber
Pattern variables	Our language	?

Figure 5: Taxonomy of languages with dynamic values

Type matching is simplest in languages like CLU [20] and Modula-2+ [33], where the construct corresponding to our `typecase` allows only exact matches (no pattern variables), and where equivalence of types is "by name." In Modula-2+, for example, the type tags of dynamic values are just unique identifiers and type matching is a check for equality.

When subtyping is involved, matching becomes more complicated. For example, Simula-67 uses name equivalence for type matching so type tags can again be represented as atoms. But to find out whether a given object's type tag matches an arm of a `when` clause (which dynamically checks whether an object's actual type is in a given subclass of its statically apparent type), it is necessary to scan the superclasses of the object's actual class. This is reasonably efficient, since the subclass hierarchy tends to be shallow and only a few instructions are required to check each level.

It is also possible to have a language with structural equivalence where type matching is still based on simple comparison of atoms. Modula-3, for example, includes a type similar to `Dynamic`, a `typecase` construct that allows only matching of complete type expressions (no pattern variables), and a notion of subtyping [8, 9]. (We do not know of a language with structural equivalence, `Dynamic`, and exact type matching, but *without* subtyping.) Efficient implementation of `typecase` is possible in Modula-3 because the rules for structural matching of subtypes are "rigid"—subtyping is based on an explicit hierarchy. Thus, a unique identifier can still be associated with each equivalence class of types, and, as in Simula-67, *match* can check that a given tag is a subtype of a `typecase` guard by quickly scanning a precompiled list of superclasses of the tag.

Amber's notion of "structural subtyping" [7] requires a more sophisticated representation of type tags. The subtype hierarchy is not based on explicit declarations, but on structural similarities that allow one type to be safely used wherever another is expected. (For example, a record type with two fields `a` and `b` is a subtype of another with just the field `a`, as long as the type of `a` in the first is a subtype of the type of `a` in the second.) This means that the set of supertypes of a given type cannot be precomputed by the compiler. Instead, `Dynamic` values must be tagged with the entire structural representation of their types—the same representation that the compiler uses internally for typechecking. (In fact, because the Amber compiler is bootstrapped, the representations are *exactly* the same.) The *match* function must compare the structure of the type tag with that of each type pattern.

The language described in this paper also requires a structural representation of types—not because of subtyping, but because of the pattern variables in `typecase` guards. In order to determine whether there is a substitution of type expressions for pattern variables that makes a given pattern equal to a given type tag, it is necessary to actually match the two structurally, filling in bindings for pattern variables from the corresponding subterms in the type tag. This is exactly the "first-order matching" problem. We can imagine speeding up this structural matching of type expressions by precompiling code to match an unknown expression against a given known expression, using techniques familiar from compilers for ML [21].

The last box in figure 5 represents an open question: Is there a sensible way to combine some notion of subtyping with a `typecase` construct that includes pattern variables? The problems here are quite similar to those that arise in combining subtyping with polymorphism (for example, the difficulties in finding principal types).

8 Conclusions

Dynamic typing is necessary for embedding a statically typed language into a dynamically typed environment, while preserving strong typing. We have explored the syntax, operational semantics, and denotational semantics of a typed lambda-calculus with the type `Dynamic`. We hope that after a long but rather obscure existence, `Dynamic` may become a standard programming language feature.

Acknowledgements

We are grateful for the insightful comments of Cynthia Hibbard, Jim Horning, Bill Kalsow, Greg Nelson, and Ed Satterthwaite on earlier versions of this paper, and for Jeanette Wing's clarification of CLU's dynamically typed values.

References

[1] Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *Computing Surveys*, 19(2):105–190, June 1987.

[2] Malcolm P. Atkinson and Ronald Morrison. Polymorphic names and iterations. Draft article, September 1987.

[3] H. P. Barendregt. *The Lambda Calculus.* North Holland, revised edition, 1984.

[4] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[5] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *Simula Begin.* Studentlitteratur (Lund, Sweden), Bratt Institute Fuer Neues Lerned (Goch, FRG), Chartwell-Bratt Ltd (Kent, England), 1979.

[6] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of the Third Annual Symposium on Software Development Environments (SIGSOFT'88)*, Boston, November 1988.

[7] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages.* Springer-Verlag, 1986. Lecture Notes in Computer Science No. 242.

[8] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Research report 52, DEC Systems Research Center, November 1989.

[9] Luca Cardelli, James Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 type system. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 202–212, January 1989.

[10] Luca Cardelli and David MacQueen. Persistence and type abstraction. In *Proceedings of the Persistence and Datatypes Workshop*, August 1985. Proceedings published as University of St. Andrews, Department of Computational Science, Persistent Programming Research Report 16.

[11] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[12] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[13] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, Laurent Hascoet, and Gilles Kahn. Natural semantics on the computer. Technical Report RR 416, INRIA, June 1985.

[14] Thierry Despeyroux. Typol: a formalism to implement natural semantics. Technical Report 94, INRIA, March 1988.

[15] Mike Gordon. Adding Eval to ML. Personal communication, circa 1980.

[16] Robert Harper. Introduction to Standard ML. Technical Report ECS–LFCS–86–14, Laboratory for the Foundations of Computer Science, Edinburgh University, September 1986.

[17] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts.* Cambridge University Press, 1986.

[18] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, Passau, Germany, February 1987. Proceedings published as Springer-Verlag Lecture Notes in Computer Science 247. The paper is also available as INRIA Report 601, February, 1987.

[19] Butler Lampson. A description of the cedar language. Technical Report CSL-83-15, Xerox Palo Alto Research Center, 1983.

[20] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual.* Springer-Verlag, 1981.

[21] David MacQueen. Private communication.

[22] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[23] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[24] David B. McDonald, Scott E. Fahlman, and Skef Wholey. Internal design of cmu common lisp on the IBM RT PC. Technical Report CMU-CS-87-157, Carnegie Mellon University, April 1988.

[25] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

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

[27] John Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988.

[28] Alan Mycroft. Dynamic types in ML. Draft article, 1983.

[29] Joseph M. Newcomer. Efficient binary I/O of IDL objects. *SIGPLAN Notices*, 22(11):35–42, November 1987.

[30] Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[31] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[32] John Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development.* Springer-Verlag, 1985. Lecture Notes in Computer Science No. 185.

[33] Paul Rovner. On extending Modula-2 to build large, integrated systems. *IEEE Software*, 3(6):46–57, November 1986.

[34] Justin Craig Schaffert. A formal definition of CLU. Master's thesis, MIT, January 1978. MIT/LCS/TR-193.

[35] Robert William Scheifler. A denotational semantics of CLU. Master's thesis, MIT, May 1978. MIT/LCS/TR-201.

[36] Satish R. Thatte. Quasi-static typing (preliminary report). In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 367–381, 1990.

[37] Mads Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, Computer Science Department, Edinburgh University, 1988. CST-52-88.

[38] Pierre Weis, María-Virginia Aponte, Alain Laville, Michel Mauny, and Ascánder Suárez. The CAML reference manual, Version 2.6. Technical report, Projet Formel, INRIA-ENS, 1989.