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 runtime. However, even in statically-typed languages, there is often the need to deal with data whose type cannot be known 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 is an exploration of the syntax, operational semantics, and denotational semantics of a simple language with the type Dynamic. We give examples of how dynamically-typed values might 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.

[*]Digital Equipment Corporation, Systems Research Center

[†]Computer Science Dept., 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.)

1 Introduction

Real programs inhabit a world much larger than themselves, a world that includes other programs and databases of persistent information. These present a problem for statically-typed languages: the lifetime of data objects can span many runs of a program; full static typechecking of programs that exchange data with 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.

For example, 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. But if we take strong typing seriously, this is unacceptable: when the data in the file happens *not* to be a bitmap, the result is 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, say, character strings, 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 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 runtime 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 the *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 runtime checks, but simply returns a `Dynamic`. We provide a 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`.

A number of systems already incorporate the mechanisms we have described. But so far these features have appeared in the context of complicated language designs, with at best an operational description of their meaning. No attention has been given to the more formal implications of dynamic typing, such as the problems of proving soundness of 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. We omit all proofs in this version of the paper.

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 [4], on the other hand, can be thought of as an infinite disjoint union—essentially equivalent to `Dynamic`. The Simula `INSPECT` statement allows a program to determine dynamically which subclass a value belongs to, with an `ELSE` clause for other subclasses that the program doesn't know or care about.

CLU [17] is a later language that incorporates the idea of dynamic typing in a static context. It has a type **anytype** and a **force** construct that attempts to coerce an **anytype** into an instance of a given type, raising an exception if the coercion is not possible. Cedar/Mesa [16] provides a very similar `REFANY` and `TYPECASE`. These features of Cedar were carried over directly into Modula-2+ [27] and Modula-3 [7,6]. In CLU and Cedar/Mesa, the primary motivation for including a dynamic type was to support programming idioms from LISP.

ML [13,21] 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 with much persistent data. Still, there have been various proposals for extending ML with a dynamic type. Gordon seems to have thought of it first [12]; his ideas were later extended by Mycroft [22]. The innovation of allowing pattern variables in **typecase** expressions (see below) seems to originate with Mycroft. Unfortunately, neither of these proposals were published.

Amber [5], 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 [2,1,8].

3 Programming with Dynamic

This section introduces the notation used in the rest of the paper—essentially Church's simply-typed lambda-calculus [9,14] with a call-by-value reduction scheme [24], 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 ⇒ 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 type-checking separately, we define the behavior of our evaluator over all terms—not just over well-typed terms. (In a compiler for this language, the type-checking 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 runtime type errors like trying to apply a number as if it were a function. The result of such computations is the distinguished value wrong:

```
5+true                  ⇒   wrong
(λz:Nat.0) (5+true)     ⇒   wrong
```

Note that in the second example a runtime 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 runtime type errors.

To make the examples in this section more interesting we also use strings, cartesian products, and recursive λ-expressions . (These 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 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
  ...
  $(\vec{X}_i)$ $(x_i:T_i)$ 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 in X_i are binding and have scope over the whole branch (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 [22]. Or we might try to choose the "most specific" pattern from the ones that match a given type tag.)

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

```
rec(print: 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)
        "<"
        || print(dynamic fst(v):X)
        || ","
        || print(dynamic snd(v):Y)
        || ">"
      (v: Dynamic)
        "dynamic " || print(v)
      else "<unknown>"
    end
```

The case for pairs illustrates a subtle point. It uses a pattern to match *any* pair, and then calls the print function recursively to print the components. To do this, it must package them into new dynamic values by tagging them with their types (it can do this because the variables X and Y are bound at runtime to the appropriate types).

Since the type tag is part of the runtime representation of a dynamic value, the case for Dynamic should probably print not only the tagged value but the type tag itself. This is straightforward, using an auxiliary function printtype with the same structure as print.

The reader may have noticed that that print doesn't do its job completely: when it 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 even precludes using printtype to show the domain and codomain types of the function, since the argument to the printtype function must be a value, not just a disembodied type.

It would be possible at this point 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 [5] and Cedar/Mesa [16] 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 interest of Dynamic lies in the interaction between the 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 still could not be affected without giving up static typechecking.

Another nice 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.)

```
λf:Dynamic. λe:Dynamic.
  typecase f of
    (X,Y) (f: X → Y)
        typecase e 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+ [27]), the set of type tags involved in a computation cannot be computed statically: our dynamic expressions can cause the creation of new tags at runtime. 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:

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

It is easy to see that the type tag on the dynamic value returned by this function must be constructed at runtime, 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

[14, 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, when applied to a function `f`, returns a fixpoint of `f`:

$$\texttt{fix f} = \texttt{f(fix f)}.$$

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

$$\texttt{(fix f)a} = \texttt{f(fix f)a}$$

One function with this property (a call-by-value version of the standard $\mathbf{Y}$ combinator [24] [3, p. 131]), can be expressed in an untyped variant of our notation as follows:

$$\texttt{fix} = \lambda\texttt{f. d d}$$

where

$$\texttt{d} = \lambda\texttt{x. }\lambda\texttt{z. (f (x x)) z.}$$

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 $\texttt{T}\rightarrow\texttt{U}$ we define $\texttt{fix}_{\texttt{T}\rightarrow\texttt{U}}$, whose type is $\texttt{((T}\rightarrow\texttt{U)}\rightarrow\texttt{(T}\rightarrow\texttt{ U))}\rightarrow\texttt{(T}\rightarrow\texttt{U)}$. Unfortunately, there is no way to just fill in suitable type declarations in the untyped `fix` given above. We need to build $\texttt{fix}_{\texttt{T}\rightarrow\texttt{U}}$ in a more roundabout way.

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

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

Next, we build an "embedding" function from each type `T` into `Dynamic`, and a corresponding "projection" function from `Dynamic` to `T`:

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

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

Now we are ready to construct $\texttt{fix}_{\texttt{T}\rightarrow\texttt{U}}$. Let

$$\begin{aligned}
\texttt{emb} &= \texttt{emb}_{\texttt{Dynamic}\rightarrow\texttt{(T}\rightarrow\texttt{U)}}\\
\texttt{proj} &= \texttt{proj}_{\texttt{Dynamic}\rightarrow\texttt{(T}\rightarrow\texttt{U)}}
\end{aligned}$$

Let

$$\begin{aligned}
\texttt{d} = \lambda\texttt{x:Dynamic. }&\lambda\texttt{z:T.}\\
&\texttt{f ((proj x) x) z}
\end{aligned}$$

(To see that this expression is well-typed, assume that `f` has type $\texttt{(T}\rightarrow\texttt{U)}\rightarrow\texttt{(T}\rightarrow\texttt{U)}$. The type of `d` works out to be $\texttt{Dynamic}\rightarrow\texttt{(T}\rightarrow\texttt{U)}$.) Finally,

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

has type $\texttt{((T}\rightarrow\texttt{U)}\rightarrow\texttt{(T}\rightarrow\texttt{U))}\rightarrow\texttt{(T}\rightarrow\texttt{U)}$, 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*:

```
T  ::=  Nat
     |  X
     |  T → U
     |  Dynamic
```

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*.

$$\begin{aligned}
\texttt{e}\ ::=\ &\texttt{x}\\
|\ &\texttt{wrong}\\
|\ &\lambda\texttt{x:T.}\texttt{e}_{\texttt{body}}\\
|\ &\texttt{e}_{\texttt{fun}}\texttt{(}\texttt{e}_{\texttt{body}}\texttt{)}\\
|\ &\texttt{0}\\
|\ &\texttt{succ }\texttt{e}_{\texttt{nat}}\\
|\ &\texttt{test }\texttt{e}_{\texttt{nat}}\texttt{ 0:}\texttt{e}_{\texttt{zero}}\texttt{ succ(x):}\texttt{e}_{\texttt{succ}}\\
|\ &\texttt{dynamic }\texttt{e}_{\texttt{body}}\texttt{:T}\\
|\ &\texttt{typecase }\texttt{e}_{\texttt{sel}}\texttt{ of}\\
&\quad\dots\\
&\quad(\vec{\texttt{X}}_\texttt{i})\ \texttt{(}\texttt{x}_\texttt{i}\texttt{:}\texttt{T}_\texttt{i}\texttt{)}\ \texttt{e}_\texttt{m}\\
&\quad\dots\\
&\quad\texttt{else }\texttt{e}_{\texttt{else}}\\
&\texttt{end}
\end{aligned}$$

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

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

Figure 1: Summary of Definitions

This is a simpler language than we used in the examples. We have omitted strings, 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 [19].

We give special names to certain subsets of *TExp* and *OpenExp*. Let $FTV(\mathtt{e})$ be the set of free type variables in e and $FV(\mathtt{e})$ the set of free variables in e. *ClosedExp* denotes the closed expressions, *Exp* denotes the type-closed expressions (expressions with no free type variables, but possibly with free variables), and *TypeCode* denotes the closed type expressions. When we write simply "expression" in this paper, we mean a type-closed expression.

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 $Value \subseteq ClosedExp$ 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 $\mathtt{e}_{\mathtt{body}}$ is in canonical form and different from `wrong` and T is closed.

A substitution σ of type variables 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. $Subst_{\vec{X}_i}$ denotes the substitutions whose domain is $\vec{X}_i$. We use a similar notation for substitution of canonical expressions for free variables.

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

We consistently use certain variables to range over particular classes of objects. The letters x, y, and z are used as metavariables ranging over variables in the language. (They are sometimes also 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.

4.2 Typechecking

Our notation for describing typechecking (and, in the next section, evaluation) is a form of "structural operational semantics" [25,19]. 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 called "natural semantics."

The rules closely follow the term structure of expressions To compute a type for $\mathtt{e}_{\mathtt{fun}}(\mathtt{e}_{\mathtt{arg}})$, for example, we first attempt to compute types for its subterms $\mathtt{e}_{\mathtt{fun}}$ and $\mathtt{e}_{\mathtt{arg}}$ 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 [13]. A group at INRIA has built a system for directly interpreting formal specifications written in a similar notation [15,10,11].

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

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

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.

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

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.

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

The constant 0 has type Nat.

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

A succ expression has type Nat if its body does.

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

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.)

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

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

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

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

The second premise is quantified over *all* substitutions $\sigma \in Subst_{\vec{X}_i}$. 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.

$$\frac{\begin{array}{c} TE \vdash \mathtt{e}_{\mathtt{sel}} : \mathtt{Dynamic} \\ \forall i, \forall \sigma \in Subst_{\vec{X}_i}.\ TE[\mathtt{x}_i \leftarrow \mathtt{T}_i\sigma] \vdash e_i\sigma : \mathtt{T} \\ TE \vdash \mathtt{e}_{\mathtt{else}} : \mathtt{T} \end{array}}{\begin{array}{l} TE \vdash (\mathtt{typecase}\ \mathtt{e}_{\mathtt{sel}}\ \mathtt{of} \\ \qquad \ldots (\vec{\mathtt{X}}_i)\ (\mathtt{x}_i{:}\mathtt{T}_i)\ \mathtt{e}_i \ldots \\ \qquad \mathtt{else}\ \mathtt{e}_{\mathtt{else}} \\ \quad \mathtt{end}) \qquad\qquad : \mathtt{T} \end{array}}$$

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 exactly the same notation as the typechecking rules. We define the judgement "closed expression e reduces to canonical expression v" (written "e $\Rightarrow$ 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 unusual 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 $\perp$ (bottom), our evaluation rules simply fail to derive any result whatsoever.

When the evaluation of an expression runs into a runtime error like trying to apply an integer 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 rules for evaluating constants are trivial, since they are already in canonical form. In particular, wrong is in canonical form.

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

Every λ-expression is in canonical form.

$$\vdash \lambda x\texttt{:}T\texttt{.}e_{body} \Rightarrow \lambda x\texttt{:}T\texttt{.}e_{body}$$

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 (i.e. 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.

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

$$\frac{\vdash e_{fun} \Rightarrow w \quad (w \neq (\lambda x\texttt{:}T\texttt{.}e_{body}))}{\vdash e_{fun}(e_{arg}) \Rightarrow \texttt{wrong}}$$

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

The constant 0 is in canonical form.

$$\vdash 0 \Rightarrow 0$$

A succ expression is in canonical form when its body is a canonical number. Therefore, 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.

$$\frac{\vdash e_{nat} \Rightarrow v \quad (v \text{ a canonical number})}{\vdash \texttt{succ}\ e_{nat} \Rightarrow \texttt{succ}\ v}$$

$$\frac{\vdash e_{nat} \Rightarrow v \quad (v \text{ not a canonical number})}{\vdash \texttt{succ}\ e_{nat} \Rightarrow \texttt{wrong}}$$

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.

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

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

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

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

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

$$\frac{\vdash e_{body} \Rightarrow \texttt{wrong}}{\vdash (\texttt{dynamic}\ e_{body}\texttt{:}T) \Rightarrow \texttt{wrong}}$$

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 runtime typecode T against a pattern expression U with free variables. If there is a substitution σ such that $T=U\sigma$, then *match*(T, U)=σ. (For the very 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 runtime environments by immediately substituting the bound variable x_i and pattern variables T_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.

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

$$\frac{\begin{array}{c}\vdash \mathtt{e}_{sel} \Rightarrow \mathtt{dynamic\ w{:}T} \\ \forall k.\ match(\mathtt{T},\ \mathtt{T}_k)\ \text{fails} \\ \vdash \mathtt{e}_{else} \Rightarrow \mathtt{v}\end{array}}{\begin{array}{l}\vdash \mathtt{(typecase\ e}_{sel}\ \mathtt{of} \\ \quad \ldots (\vec{X}_i)\ (\mathtt{x}_i{:}\mathtt{T}_i)\ \mathtt{e}_i \ldots \\ \quad \mathtt{else\ e}_{else}) \\ \mathtt{end)} \qquad\qquad \Rightarrow \mathtt{v}\end{array}}$$

$$\frac{\vdash \mathtt{e} \Rightarrow \mathtt{v}\quad (\mathtt{v} \neq \mathtt{(dynamic\ w{:}T)})}{\begin{array}{l}\vdash \mathtt{(typecase\ e}_{sel}\ \mathtt{of} \\ \quad \ldots (\vec{X}_i)\ (\mathtt{x}_i{:}\mathtt{T}_i)\ \mathtt{e}_i \ldots \\ \quad \mathtt{else\ e}_{else}) \\ \mathtt{end)} \qquad\qquad \Rightarrow \mathtt{wrong}\end{array}}$$

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 be assigned the same type as e by the static semantics. From this it is an easy corollary that no well-typed program can evaluate to wrong.

Theorem 4.4.1 (Soundness) *For all expressions* e, *values* v, *and types* T, *if* $\vdash$ e $\Rightarrow$ v *and* $\vdash$ e : T, *then* $\vdash$ v : T.

Since wrong is not assigned any type by the static semantics, the following is immediate:

Corollary 4.4.2 *For all expressions* e, *canonical expressions* v, *and types* T, *if* $\vdash$ e $\Rightarrow$ v *and* $\vdash$ e : T *then* v $\neq$ wrong.

5 Denotational Semantics

Another way of showing that our typing system is 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 $[\![\mathtt{wrong}]\!]_\rho$ = wrong for all $\rho \in Var \rightarrow \mathbf{V}$.

Two properties are highly desirable:

- If e is a well-typed expression then $[\![\mathtt{e}]\!]_\rho \neq$ wrong for all ρ.
- If $\vdash$ e $\Rightarrow$ 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 **V** not containing wrong, and prove:

- If $\vdash$ e : 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 **V** and to typecodes as subsets of **V**. We prove that the evaluation and typechecking rules are sound with respect to the semantics.

In order to give a meaning to typecodes, we use the ideal model of types, following MacQueen, Plotkin, and Sethi [18]. (We refer the reader to this paper for the technical background of our construction.) Typecodes denote ideals—nonempty subsets of **V** closed under approximations and limits. We denote by **Idl** the set of all ideals in **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 **V** that satisfies the isomorphism equation

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

where **N** is the flat domain of natural numbers, **W** is the type-error domain $\{w\}_\perp$. The usual continuous function space operation is represented as $\rightarrow$; the product-space $E \times A$ of a cpo E and a set A is defined

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

$$
\begin{array}{lcl}
\llbracket \mathtt{x} \rrbracket_\rho & = & \rho(\mathtt{x}) \\
\llbracket \mathtt{wrong} \rrbracket_\rho & = & \mathbf{wrong} \\
\llbracket \lambda \mathtt{x}:\mathtt{T}.\mathtt{e}_{\mathtt{body}} \rrbracket_\rho & = & (\lambda v.\ \text{if } v = \mathbf{wrong} \text{ then } \mathbf{wrong} \text{ else } \llbracket \mathtt{e}_{\mathtt{body}} \rrbracket_{\rho\{x \leftarrow v\}}) \text{ in } \mathbf{V} \\
\llbracket \mathtt{e}_{\mathtt{fun}}(\mathtt{e}_{\mathtt{arg}}) \rrbracket_\rho & = & \text{if } \llbracket \mathtt{e}_{\mathtt{fun}} \rrbracket_\rho \not\sqsubset (\mathbf{V} \to \mathbf{V}) \text{ then } \mathbf{wrong} \text{ else } (\llbracket \mathtt{e}_{\mathtt{fun}} \rrbracket_\rho |_{\mathbf{V} \to \mathbf{V}})(\llbracket \mathtt{e}_{\mathtt{arg}} \rrbracket_\rho) \\
 & & \\
\llbracket \mathtt{0} \rrbracket_\rho & = & 0 \text{ in } \mathbf{V} \\
\llbracket \mathtt{succ}\ \mathtt{e}_{\mathtt{nat}} \rrbracket_\rho & = & \text{if } \llbracket \mathtt{e}_{\mathtt{nat}} \rrbracket_\rho \not\sqsubset \mathbf{N} \text{ then } \mathbf{wrong} \text{ else } (\llbracket \mathtt{e}_{\mathtt{nat}} \rrbracket_\rho |_{\mathbf{N}} + 1) \text{ in } \mathbf{V} \\
\multicolumn{3}{l}{\llbracket \mathtt{test}\ \mathtt{e}_{\mathtt{nat}}\ \mathtt{0}:\mathtt{e}_{\mathtt{zero}}\ \ \mathtt{succ(x)}:\mathtt{e}_{\mathtt{succ}} \rrbracket_\rho} \\
 & = & \text{if } \llbracket \mathtt{e}_{\mathtt{nat}} \rrbracket_\rho \not\sqsubset \mathbf{N} \text{ then } \mathbf{wrong} \\
 & & \text{else if } \llbracket \mathtt{e}_{\mathtt{nat}} \rrbracket_\rho = 0 \text{ in } \mathbf{V} \text{ then } \llbracket \mathtt{e}_{\mathtt{zero}} \rrbracket_\rho \\
 & & \text{else } \llbracket \mathtt{e}_{\mathtt{succ}} \rrbracket_{\rho\{x \leftarrow ((\llbracket \mathtt{e}_{\mathtt{nat}} \rrbracket_\rho |_{\mathbf{N}} - 1) \text{ in } \mathbf{V})\}} \\
 & & \\
\llbracket \mathtt{dynamic}\ \mathtt{e}_{\mathtt{body}}:\mathtt{T} \rrbracket_\rho & = & \text{if } \llbracket \mathtt{e}_{\mathtt{body}} \rrbracket_\rho = \mathbf{wrong} \text{ then } \mathbf{wrong} \text{ else } (\langle \llbracket \mathtt{e}_{\mathtt{body}} \rrbracket_\rho, \mathtt{T} \rangle \text{ in } \mathbf{V}) \\
\multicolumn{3}{l}{\llbracket \mathtt{typecase}\ \mathtt{e}_{\mathtt{sel}}\ \mathtt{of}\ \ldots (\vec{X_i})\,(\mathtt{x}_i:\mathtt{T}_i)\,\mathtt{e}_i \ldots \mathtt{else}\ \mathtt{e}_{\mathtt{else}} \rrbracket_\rho} \\
 & = & \text{if } \llbracket \mathtt{e}_{\mathtt{sel}} \rrbracket_\rho \not\sqsubset (\mathbf{V} \times \mathit{TypeCode}) \text{ then } \mathbf{wrong} \\
 & & \text{else let } \langle d, \mathtt{U} \rangle = \llbracket \mathtt{e}_{\mathtt{sel}} \rrbracket_\rho |_{\mathbf{V} \times \mathit{TypeCode}} \text{ in} \\
 & & \quad \text{if } \ldots \\
 & & \quad \text{else if } \mathit{match}(\mathtt{U}, \mathtt{T}_i) \text{ succeeds} \\
 & & \qquad \text{then let } \sigma = \mathit{match}(\mathtt{U}, \mathtt{T}_i) \text{ in } \llbracket \mathtt{e}_i \sigma \rrbracket_{\rho\{x_i \leftarrow d\}} \\
 & & \quad \text{else if } \ldots \\
 & & \quad \text{else } \llbracket \mathtt{e}_{\mathtt{else}} \rrbracket_\rho
\end{array}
$$

Figure 2: The Meaning Function for Expressions

as $\{\langle e, a\rangle \mid e \in E,\ e \neq \bot, \text{ and } a \in A\} \cup \{\bot_E\}$, with the evident ordering.

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

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

We omit the details of the construction, which are standard [3,18].

At this point, we have a universe suitable for assigning a meaning to expressions in our programming language. Figure 2 gives a full definition of the denotation function $\llbracket\ \rrbracket$, with 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 $\mathbf{V})$;
- $v|_{\mathbf{S}}$ yields: if $v = (d$ in $\mathbf{V})$ for some $d \in \mathbf{S}$ then d, and $\bot$ otherwise;
- $v \sqsubset \mathbf{S}$ yields $\bot$ if $v = \bot$, true if $v = (d$ in $\mathbf{V})$ for some $d \in \mathbf{S}$, and false otherwise;
- $=$ yields $\bot$ whenever either argument does.

Evaluation is sound with respect to the denotation function:

Theorem 5.1.1 *For all expressions* e *and* v, *if* $\vdash \mathtt{e} \Rightarrow \mathtt{v}$ *then* $\llbracket \mathtt{e} \rrbracket = \llbracket \mathtt{v} \rrbracket$.

Although we now have a meaning $\llbracket \mathtt{e} \rrbracket$ for each program e, we do not yet have a meaning $\llbracket \mathtt{T} \rrbracket$ for each typecode 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 5.1.2 *If $I \subseteq \mathbf{V}$ is a set of values and* 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 5.1.3 *If $I \subseteq \mathbf{V}$ and $J \subseteq \mathbf{V}$ are two sets of values, then*

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

Note that if I and J are ideals then so is $I \longrightarrow\!\!\!\!\!\rightarrow J$.

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

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

Here the type 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 $\langle c, \mathtt{T} \rightarrow \mathtt{U} \rangle$ 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

$$\begin{array}{l} d(G(X_1, \ldots, X_n), G(X'_1, \ldots, X'_n)) \\ \qquad \leq t \cdot max\{d(X_i, X'_i) \mid 1 \leq n\}. \end{array}$$

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

Theorem 5.1.4 *The operation $\times$ is contractive (when its second argument is fixed). The operation $\longrightarrow\!\!\!\!\!\rightarrow$ is contractive.*

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

Theorem 5.1.5 *The equation*

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

has a unique solution in **Idl**.

Let us call this solution **Dynamic**.

$$\boxed{\begin{array}{lcl} \multicolumn{3}{l}{[\![\ ]\!] : \mathit{TypeCode} \rightarrow \mathbf{Idl}} \\ & & \\ [\![\mathtt{Nat}]\!] & = & \mathbf{N} \\ [\![\mathtt{Dynamic}]\!] & = & \mathbf{Dynamic} \\ [\![\mathtt{T} \rightarrow \mathtt{U}]\!] & = & \{c \mid c([\![\mathtt{T}]\!]) \subseteq [\![\mathtt{U}]\!]\} \end{array}}$$

Figure 3: The Meaning Function for Typecodes

Finally, we are in a position to associate an ideal $[\![\mathtt{T}]\!]$ with each typecode T (see figure 3). The semantics fits our original intuition of what dynamic values are, as the following proposition shows.

Proposition 5.1.6 *For all values v and typecodes* T*, $\langle v, \mathtt{T} \rangle \in$ **Dynamic** if and only if $v \in [\![\mathtt{T}]\!]$.*

We can also prove the soundness of typechecking:

Definition 5.1.7 *The environment ρ is consistent with the type environment TE on the expression* e *if $TE(\mathtt{x})$ is defined and $\rho(\mathtt{x}) \in [\![TE(\mathtt{x})]\!]$ for all $\mathtt{x} \in FV(\mathtt{e})$.*

Theorem 5.1.8 *For all type environments TE, expressions* e*, environments ρ consistent with TE on* e*, and typecodes* T*, if $TE \vdash \mathtt{e} : \mathtt{T}$ then $[\![\mathtt{e}]\!]_\rho \in [\![\mathtt{T}]\!]$.*

It follows from Theorem 5.1.1, Theorem 5.1.8, and the fact that no $[\![\mathtt{T}]\!]$ can contain (w in $\mathbf{V}$) that no well-typed expression evaluates to **wrong**.

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 which assign 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

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

A similar use of sums appears in Mycroft's work [22].

6 Polymorphism

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. For most of the section, we assume an explicitly-typed polymorphic lambda calculus along the lines of Reynolds' system [26]. The type abstraction operator is written as Λ. Type application is written with square brackets.

Our motivating example 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. Now 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 (written with square brackets to distinguish them from ordinary pattern variables: `X[]`) range over functions from type expressions to type expressions.

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

```
λf:Dynamic. λe:Dynamic.
  typecase f of
    (X[],Y[]) (f: ∀(Z) X[Z] → Y[Z])
      typecase e of
        (W) (e: X[W])
          dynamic f(W)(e):Y[W]
        else
          dynamic "Error":String
      end
    else
      dynamic "Error":String
  end
```

For instance, when we apply the function to the arguments

```
f = dynamic (Λ Z.λx:Z.x): (∀(Z) Z→Z)
e = dynamic 3:Nat
```

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

An even more intriguing example is polymorphic dynamic composition. To check that two parameters are both polymorphic functions and that their composition is well-typed, returning the composition if so, we might write something like this:

```
λf:Dynamic. λg:Dynamic.
  typecase f of
    (X,Y) (f: ∀(W) X[W]→ Y[W])
      typecase g of
        (Z) (g: ∀(V) Y[V]→ Z[V])
          dynamic (Λ W. g[W] ∘ f[W])
                      : ∀V.X[V]→ Z[V]
        else ...
      end
    else ...
  end
```

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

```
Λ X. λx:X. dynamic x:X
```

The extra cost of actually performing type abstractions and applications at runtime (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 [22] 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 monotype or else totally undetermined (i.e., 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 runtime. Dynamics with non-generic variables can be used to break the ML type system. (The problem is analogous to that of "updateable refs" [28].)

This preliminary treatment of polymorphism 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.

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

Figure 4: Taxonomy of languages with dynamic values

explicit polymorphism? We hope that our two 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, the implementation of the *match* function, and the 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 runtime 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 [20] for a description of a typical fasl format. The idea goes back to at least 1974.)

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 DEC Systems Research Center. Another scheme was implemented as part of Tartan Labs' Interface Description Language [23]. 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 language have existed for some time (see section 2). Figure 4 gives a rough classification of several languages according to the amount of work involved in comparing types and the presence or absence of subtyping.

Type matching is simplest in languages like CLU [17] and Modula-2+ [27], where the construct corresponding to our **typecase** allows only exact matches (no pattern variables), and where equivalence of types is "by name." In such languages, the type tags of dynamic values are simply unique identifiers and type matching is just 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 [6,7]. (Interestingly, 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, *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" [5] 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. 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 techinques familiar from compilers for ML.

The last box in figure 4 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 Bill Kalsow, Greg Nelson, and Ed Satterthwaite on earlier versions of this paper.

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. September 1987. Draft article.

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

[4] 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.

[5] 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.

[6] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. *Modula-3 Report*. Technical Report 31, DEC Systems Research Center, 1988.

[7] 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*, January 1989.

[8] 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.

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

[10] Dominique Clément, Joëlle Despeyroux, Thierry Despeyrous, and Gilles Kahn. A simple applicative language: mini-ML. Draft article (INRIA).

[11] Joëlle Despeyroux. Proof of translation in natural semantics. Draft article (INRIA).

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

[13] Robert Harper, Robin Milner, and Mads Tofte. *The Semantics of Standard ML: Version 1.* Technical Report ECS-LFCS-87-36, Computer Science Department, University of Edinburgh, 1987.

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

[15] Gilles Kahn, Dominique Clément, Joëlle Despeyroux, Thierry Despeyrous, and Laurent Hascoet. Natural semantics on the computer. Draft article (INRIA).

[16] Butler Lampson. *A Description of the Cedar Language.* Technical Report CSL-83-15, Xerox Palo Alto Research Center, 1983.

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

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

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

[20] 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.

[21] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 1978.

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

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

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

[25] Gordon D. Plotkin. *A Structural Approach to Operational Semantics.* Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

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

[27] Paul Rovner, Roy Levin, and John Wick. *On Extending Modula-2 for Building Large, Integrated Systems.* Technical Report Technical Report 3, Digital Systems Research Center, 1985.

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