Compiling dynamic languages via typed functional languages

Raj Bandyopadhyay
Rice University
rajb@rice.edu

Walid Taha
Rice University
taha@rice.edu

Abstract
Dynamic languages enable rapid prototyping, while statically typed languages offer early error-detection and efficient execution. As a result, the usual practice in software development is to build a prototype in a dynamic language and then rewrite the application in C or Fortran for high performance. Our thesis is that this costly rewriting step can be avoided if we have good native code compilers for dynamic languages.

To overcome the difficulties in building good native code compilers from scratch, we propose that dynamic languages can be compiled into high-performance native code executables by translating to typed functional languages and reusing existing functional language compilers. We demonstrate this approach by compiling a popular dynamic language, Python, by translating it to OCaml, a strongly typed functional language. On performing a comparative evaluation against several available Python implementations on both Windows and Linux platforms, we obtain highly encouraging results.

In this paper, we use Python as proof-of-concept to demonstrate that our approach delivers efficient native code compilers for dynamic languages. We describe how source dynamic language objects and constructs can be expressed in terms of target typed functional language data types. Finally, we present a comparative performance analysis against different Python implementations such as CPython, IronPython, PyPy and Jython to illustrate the effectiveness of our approach.

1. Introduction
Dynamic scripting languages have become ubiquitous in a wide range of programming domains, including scientific computing. Some of these languages which are widely used include Python, Perl, Matlab and R. These languages provide increased flexibility and improve productivity by providing domain-specific features and abstracting many of the details associated with lower-level programming languages such as C and Fortran.

On the other hand, statically typed languages offer a slew of advantages including higher safety, early error detection and higher execution efficiency. As a result, the standard practice is to use dynamic languages for prototyping and use a lower-level language to ‘harden’ the application for performance. Our thesis is that we can avoid this costly rewriting step if we have good native code compilers for dynamic languages.

In this paper, we first describe some of the challenges in compiling dynamic languages in general, and Python in particular. We then describe an abstract syntax for our source and target languages. The following section describes the translation process, including features implemented in runtime environment, the architecture of our compiler and the translation semantics for specific language constructs. We then perform a comparative analysis of our implementation against other Python implementations and discuss the effectiveness of our approach.

Writing a good native code compiler from scratch for each dynamic language is hard. In addition to language and platform-specific idiosyncrasies, dynamic languages present several challenges that make them harder to compile compared to statically typed languages. These include automatic memory management and the lack of static typing information. In addition, many common scripting languages such as Python, Perl and R have evolving semantics, or the language semantics is not well-defined enough for formal reasoning, making the compiler developer’s task even harder.

To overcome these difficulties, we propose to develop good native code compilers for dynamic languages by translating them to a strongly typed functional languages as intermediate. Using typed functional languages allows us to utilize the large body of existing infrastructure developed by the functional programming community, both in theoretical research and practical tools. This includes efficient native code compilers and memory management runtimes.

We demonstrate the effectiveness of our approach by developing a compiler for a popular dynamic language, Python using a strongly typed functional language, OCaml as intermediate. Python is a highly dynamic object-oriented language. The standard Python implementation, CPython, is a bytecode interpreter written in C. We present a comparative evaluation of our implementation against several available Python implementations such as IronPython, PyPy and Jython on Windows and Linux platforms.

The technical contributions of this paper are:

- A methodology for building native code compilers for dynamic languages using statically typed functional languages,
- A formal representation of the source dynamic language (Python), target functional language (OCaml), and the translation semantics,
- A representation of source dynamic language objects in terms of target functional language data types,
- A native code compiler for Python via translation to OCaml as proof-of-concept,
- A detailed comparative performance evaluation of our implementation versus several available Python implementations on multiple platforms.

In this paper, we first describe some of the challenges in compiling dynamic languages in general, and Python in particular. We then describe an abstract syntax for our source and target languages. The following section describes the translation process, including features implemented in runtime environment, the architecture of our compiler and the translation semantics for specific language constructs. We then perform a comparative analysis of our implementation against other Python implementations and discuss the effectiveness of our approach.
2. Challenges in compiling dynamic languages

Dynamic languages such as Python, Perl, Matlab and R present some common challenges for language developers.

- **Memory Management** Most scripting languages abstract the intricacies of memory allocation and deallocation away from the user. This improves runtime safety by automating the responsibility of freeing allocated memory chunks as needed. However, this necessitates a larger amount of work performed by the runtime environment.

- **Lack of compile-time type information** Having type information at compile time enables the compiler to optimize memory allocation and reduce overhead of dynamic method lookup. Dynamic languages usually have no type declarations, and types can be coerced to one another at runtime making precise type inference hard.

- **Imprecise definition of semantics** Scripting languages such as Python and Perl have been developed cooperatively over a period of years, incorporating features and code developed by many users. As a result, many features of the language and their behavior in corner cases are not always well documented. In some cases, behaviors are left undefined or implementation-dependent.

2.1 Benefits of using typed functional languages

The most important benefit of using a typed functional language as an intermediate is that these languages provide highly efficient automatic memory management and garbage collection. Any language that is effectively translated to a functional language can immediately take advantage of these facilities.

The type systems of functional languages such as ML and Haskell are expressive enough to model both untyped and fully dynamic computation as well as highly precise typed values. This is done using a combination of primitive types, structures such as tuples and records, and tagged unions with pattern-matching facilities over tags.

Functional languages have an extremely well-defined and well-understood semantics. Translating to a functional language gives us a formal semantics for the source language in the form of the translator.

In this research, we use a strongly typed functional language, OCaml (?), both as our intermediate representation of code as well as our compiler development language. Compiling a dynamic language to an ML-like language such as OCaml has several novel benefits:

- The OCaml implementation is standardized and has a competitive, stable and widely used native code compiler
- OCaml has a foreign function interface that is well-integrated with the language
- Translating to a higher-level language is easier, and allows the compiler writer to focus on the most effective optimizations rather than the myriad of details to be managed while translating to a lower-level language
- OCaml provides type inference, allowing the compiler writer to express translations without having to generate explicit typing annotations
- In addition, OCaml is a functional-imperative language, providing some imperative features such as references, assignments and statement sequences, which allows effective expression of constructs in most dynamic scripting languages.

2.2 Garbage Collection

Automatic memory management is implemented in most scripting languages for higher productivity. The CPython implementation uses a strategy called *Reference Counting*. Every object contains a reference count, which is incremented or decremented based on whether a reference to the object is added or removed in the program. When the reference count becomes zero, the object’s memory can be freed.

The main advantages of reference counting are that it is fairly easy to implement and highly portable. It only requires that the functions `malloc()` and `free()` be available, which is guaranteed by the C standard. Therefore, it is used in scripting languages such as Python and Perl, which are implemented in C. The chief drawback of reference counting is that it cannot be used in the presence of cycles, when an object refers to itself (usually indirectly). Python offers an additional optional `gc` module to manage memory in the presence of cycles. In addition, reference counts take up extra memory and need to be updated correctly.

Automatic garbage collection algorithms not only manage allocation and deallocation correctly, but are also more efficient in many cases due to their use of *heap compaction* and better cache performance. The most common algorithms are known as *generational* collectors, which are implemented both in modern JVMs, as well as in functional languages such as OCaml. Most generational collectors split the heap into a *young* and *old* generation (minor and major heaps in OCaml). Objects first move into the minor heap, and if they have survived a fixed number of collection cycles, move into the major heap. In addition, OCaml’s garbage collector is *incremental* i.e. it interleaves collection with computation, thus maintaining performance.

2.3 Challenges in Python

The following Python script illustrates many of the syntactic features of Python as well as the challenges it presents in compilation. The program creates a new class *MyInt* by inheriting from the built-in *int* and overrides the default *+* operator to compute the sum modulo 2.

```python
glob = 2  # a global variable
class MyInt(int):
    # Constructor function
    def __init__(self, x = 0):
        self.v = x

    # The '+' operator for this type
    def __add__(self, x):
        return (self.v + x)%glob

i = MyInt(5)  # instantiate
print i + 10  # prints 1
print i * 20  # prints 100
```

Figure 1. A simple Python example

- **Customizable object behavior** Python objects are highly introspective and mutable. An object can examine its properties and methods at runtime, and methods can be added and deleted at any point. This customization is achieved by using a mutable dictionary structure to represent the methods of every object.

- **Behavioral ("duck") typing** In Python, the behavior of an object is determined by its current set of methods and properties, rather than by inheritance from a base class. Two objects are interchangeable if they implement the same external method interface, regardless of their inheritance hierarchy.
• Function call semantics Python allows both positional and named (keyword) arguments in function calls, along with default values for parameters in function definitions. This can make function call semantics quite expensive.

• Dynamic scoping rules Python is largely lexically (statically) scoped, but allows the environment to be queried during execution. The retrieved environment objects may also be mutable, thus violating lexical scoping norms. Python supports nested classes and functions, and global variables may be declared within any inner scope.

• Dynamic loading with exec and eval Python supports dynamic loading and execution of code via the exec and eval functions. This makes compile-time optimization and inference harder, since we may have little or no information about the code being loaded.

• Deletion (finalization) methods Python supports finalization methods for objects i.e. certain methods may be called when an object is freed from memory. This is a highly implementation-specific feature, and depends on the memory management system used. Under an automatic garbage collector, the order and time when objects are deallocated is not predictable, hence a program whose correctness depends on deletion methods may not work as expected.

• Foreign function interface One of the reasons for Python’s popularity is the ease with which it interfaces with several lower-level languages such as C, C++ and Fortran. The standard Python implementation is written in C and defines a standard method of interfacing with C/C++. However, these foreign function interfaces are highly implementation-dependent.

3. Python: source dynamic language

Python is a highly feature-rich language with many syntactic constructs and shortcuts. We present a core subset of Python as a formal BNF.

Program  \( p ::= \langle D_i \rangle \in E I \ m \)

Module \( D ::= \langle S_i \rangle \in E I \)

Statement \( S ::= e \mid l = e \mid \text{global} \ x \mid \langle S_i \rangle \in E I \ |
\text{if} \ e :: S \text{else} :: S \ |
\text{for} \ l \in e :: S \ |
\text{while} \ e :: S \ |
\text{return} \ e \ |
\text{yield} \ e \ |
\text{break} \ |
\text{continue} \ |
\text{pass} \ |
\text{def} \ x(x_i \in E I, (x_j = e_j) \in E I) :: S \ |
\text{class} x(x_i \in E I) :: S \ |
\text{try} :: S \text{except} :: S \ |
\text{raise} \ e \)

Expression \( e ::= e \mid x \mid e \cdot x \mid e \cdot e \ |
\langle e_i \rangle \in E I \ |
\text{[e]} \ |
\text{[e:e:e]} \ |
\langle x_i = e_i \rangle \in E I \ |
\text{e op e} \ |
\text{e(e,e(x_i = e_i) \in E I)} \)

LHS value \( l ::= x \mid e \cdot x \mid e[e] \ |
\text{[e:e:e]} \ |
\text{[x_i \in E I]} \)

Constants \( c ::= \text{[None,True,False,0,0.0,"",...]} \)

Python syntax includes built-in constants, including the special value None. Variables are generated by assignment, which could be to a name or an object attribute. In addition, Python has the standard control-flow constructs such as for and while loops, if statements, and non-local constructs such as break, continue and return statements. The standard operators, both boolean and binary are present. The try . . . except syntax handles exceptions, which are raised by raise.

4. OCaml: Target functional language

OCaml, being a strongly typed functional language, has a very simple core calculus. However, it provides many syntactic constructs which can express dynamic language constructs effectively. We describe the abstract syntax of the core subset of OCaml that we use as the target of our translation scheme.

Program \( p ::= \langle D_i \rangle \in E I \ m \)

Module \( D ::= e \)

Expressions \( e ::= () \mid e \cdot x \mid e \cdot e \mid e \ |
\langle e_i \rangle \in E I \ |
\text{[e]} \ |
\text{[e:e:e]} \ |
\langle e_i \rangle \in E I \ |
\text{if} \ e \text{ then} S \ |
\text{else} S \ |
\text{while} e \text{ do} S \ |
\text{ref} e \ |
\text{let} \ x \ ::= e \ |
\text{fun} (x_i \in E I) \rightarrow e \ |
\text{let} (x_i = e_i) \in E I \ |
\text{match} e \text{ with} \ (p_i \rightarrow e_i) \in E I \ |
\text{try} e \ |
\text{raise} e \)

Constants \( c ::= \text{[true,false,0,0.0,"",...]} \)

Patterns \( p ::= \text{Tag} \ x \)

Our core subset includes the standard constants and the () value, which has type unit. We also have variables, function application and let-definitions. In particular, function application is a highly used feature of our translation, since most Python operators and syntactic constructs are implemented as functions inside the runtime environment. We use the standard control-flow constructs such as if statements and while loops, and OCaml’s imperative facilities: references, assignments and statement sequences. Finally, we use OCaml’s exception raising and handling facilities extensively, both to model Python exceptions as well as non-local control flow.

5. Translating to a typed functional language

In this section, we describe in detail our methodology for translating a dynamic language to a typed functional intermediate. We first describe our implementation of key features of the Python runtime. We then show how specific language constructs are translated, and outline the overall design of our compiler.

5.1 Implementing key Python runtime features

We use the standard Python implementation, CPython, as our reference for Python runtime semantics. In addition, the Python Reference Manual (?) is the most important reference for a high-level overview of the internals of Python. We have implemented the runtime environment for Python, including built-in types such as int, float, list, dict and their methods in OCaml. This runtime also includes built-in functions such as map, operators such as +, and algorithms to process function arguments, build class objects and method lookup to search a class hierarchy.

Objects Everything in Python is an object. The structure and behavior of any Python object is encoded by a type object. Every object has a reference to its type object fixed at the time of its creation. Every Python object also has an unique identity and a value. The identity is implementation-dependent e.g. in CPython, it is the actual memory address of the object. Most Python objects have immutable values, except for data structures such as lists and
dictionaries. The value of an object encodes its state. The behavior of an object is described by methods in its dictionary. Since Python has multiple inheritance, the methods of an object may be located across several dictionaries in the inheritance hierarchy.

/* General Python object */
struct PyObject {
  ...
  PyTypeObject *ob_type;
}

/* Python integer object */
struct PyIntObject {
  ...
  PyTypeObject *ob_type;
  long ob_ival;
}

/* Python type object */
struct PyTypeObject {
  ...
  PyTypeObject *ob_type;
  PyObject *tp_dict;
  PyObject *tp_call;
  PyObject *tp_new;
  PyObject *tp_init;
  ...
}

Figure 2. CPython representation of Python objects

C representation CPython represents objects using C structures (Figure 2). Each C structure can be cast to the skeleton structure PyObject. Each kind of object is represented by its own C structure. For example, Python int objects are represented by PyIntObject and types are represented by the structure PyTypeObject. Type objects store a variety of information describing the object they represent. This includes the object’s size, its fields and methods. Many of these methods have their own fields. All type objects also have a dictionary tp_dict which contains these methods indexed by name.

type raw =
  | Type | Object | None_raw
  | Int of int
  | Float of float
  | String of string
  | Range of int list (* ranges and slices *)
  | Aseq of obj array (* lists and tuples *)
  | PyException of (string * dict *(obj list))
  | Ufunc of (string * ufunc * func_record)
  ... (* other values *)

and ufunc = obj -> obj -> obj

(* modified hashtable for dictionaries*)
and dict = {
  mutable size: int;
  mutable data: bucketlist array
} (* the buckets *)
and bucketlist =
  | Empty | Cons of obj * obj * bucketlist

Figure 3. OCaml tagged union for Python values

OCaml representation We use records in OCaml to represent objects. OCaml records differ from C structures in two ways. Firstly, two OCaml record types cannot share a field with the same name. Secondly, one record type cannot be cast to another. This implies that in OCaml, we have to use exactly one record type to represent all possible Python objects. In order to allow runtime modification as per Python semantics, most of the fields have to be mutable. We represent different possible value types in Python, such as integers, floats, function code etc using a tagged union type raw, as shown in Figure 3.

type obj =
  {
    obj_idx : int; (*identity*)
    loc_dict : dict option;
    mutable obj_type : obj option;
    mutable obj_value : raw;
    mutable hash_value : int;
    mutable obj_size : int;
    mutable props : tp_record option;
  }

and tp_record =
  {
    tp_name : string;
    mutable tp_bases: obj list;
    mutable tp_dict : dict ;
    mutable num_prot: numeric_protocol;
    mutable cmp_prot: compare_protocol;
    mutable tp_bool : obj -> bool; (*truth value*)
    mutable tp_hash : obj -> int; (*hash func*)
    mutable tp_call : ternaryfunc; (*callable objects *)
    mutable tp_new : ternaryfunc; (*metaclasses*)
    mutable tp_init : ternaryfunc; (* classes *)
    mutable tp_iter : unaryfunc; (*iterator*)
    mutable tp_repr : obj -> string; (*Attributes*)
    mutable tp_getattro : obj -> obj -> obj;
    mutable tp_setattro : obj -> obj -> obj -> unit;
    ...
  }

and numeric_protocol =
  {
    _add_ : obj -> obj -> obj;
    _sub_ : obj -> obj -> obj;
    _neg_ : obj -> obj;
    ...
  }

Figure 4. Representing Python objects in OCaml

As shown in Figure 4, the record for every object contains an identity, a type and a value. In addition, we store some more information; a hash value for immutable objects, the size (length) for sequence objects and an optional local dictionary for subtypes and their instances. For type objects, we need several other fields to store the standard methods. In order to efficiently utilize memory, we create another record type tp_record specifically containing these methods. These methods include functions for hashing, truth value, construction and attribute handling. They also include other important information about the type such as its name, bases and its dictionary. In addition, they include pointers to specific protocols, which are records containing methods for specific behaviors.
For instance, the numeric protocol contains methods for numeric operations. There are protocols for numbers, sequences, mappings and comparisons.

**Dictionaries**  Dictionaries in Python are mappings representing finite indexed sets of objects: \( \text{dict} : \text{obj} \rightarrow \text{obj} \). Dictionaries enable many of the dynamic features in Python. All Python objects may be used as dictionary keys, except for mutable types such as lists and dictionaries themselves. CPython implements the dictionary using a hash table data structure. Both equality and hashing are defined in Python as methods implemented within objects. For example, \( \text{o1} \stackrel{=} \text{o2} \) internally calls \( \text{o1 \_eq\_}(\text{o2}) \). Every hashable object must provide a \( \text{\_hash\_} \) method: \( \text{hash} : \text{obj} \rightarrow \text{long} \text{ integer} \). The only requirement is that two objects which compare equal must have the same hash value.

In our implementation, we have modified the OCaml `Hashtbl` standard library (Figure 3) to use the Python notion of equality and hashing. Our hash function uses the built-in OCaml `Hashtbl.hash` function internally, but applies it differently for each built-in type. In addition, for each immutable object, we compute the hash value exactly once and store it in the `hash_value` field of the object.

**Classes and Inheritance**  Classes are the Python user’s way of creating new types. Python classes are more flexible and powerful than those in other object-oriented languages such as C++ or Java due to its support for metaclasses. Classes in Python are themselves objects, and each class is an instance of a metaclass.

Up to Python 2.1, Python’s object-oriented model was similar to Java or C++ in that there was only one metaclass classobj. Types defined using these `class` classes were fundamentally different from built-in types such as `int`. Python 2.2 unified types and classes (?) by allowing new-style classes to seamlessly inherit from built-in types. It also allowed users to customize class behavior by defining their own metaclasses.

The distinction between built-in and user-defined types was primarily due to performance reasons. Built-in types are implemented in C according to the Python/C API, and are therefore much faster. However for new-style classes, there is an overhead at the interface of the two languages. In our implementation, both the runtime environment as well as the target language are the same, OCaml, hence the unification occurs naturally.

**Managing the environment**  The functions available in the environment when Python is loaded are all part of the \( \text{\_\_\_\_builtins\_\_\_\_} \) module. Python allows modules to be imported using the `import` statement. The dictionary of each module becomes part of the environment. In addition, the file or script currently being compiled has its own global dictionary which may be accessed and modified using the `globals()` function. Finally, the local variables of a class or function can be accessed using the `locals()` function.

For the global environment, we maintain a stack of dictionaries called `modStack`. The top of the stack is the current working module. This is the dictionary which the `globals()` function can access. An assignment to a global variable is translated to a modification to this dictionary. When the module is loaded it is popped off this stack and added to a queue `modQueue` of imported modules. This queue is searched in order when a variable has to be searched among imported modules.

Within a file, we maintain another stack of dictionaries, called `localStack`. This stack represents all the local scopes (classes and functions) within a module. The top of the stack is the current working scope. An item is pushed or popped off the `localStack` as the execution enters or leaves a local scope. The stacks and queues are implemented using the OCaml `Stack` and `Queue` standard libraries respectively.

**Exceptions**  Exceptions in Python are themselves types. Python defines several built-in exceptions for specific runtime errors. Examples include `AttributeError`, `IndexError` and so on. In our OCaml implementation, we define a single exception called `PythonException`, which takes a tuple of Python objects as its argument. This enables us to conveniently translate Python exceptions to OCaml exceptions.

```
exception PythonException of (obj * obj * obj)
```

Every Python exception has three components: the exception object itself, a message argument object and a stack trace object. We currently do not support the stack trace facility.

### 5.2 Translation from source to target

We now describe how our source language constructs are syntactically translated to those in the target language. The translation semantics is shown in Figure 5.2. In this section, we describe some of the translations in detail.

**Constants**  The basic translation semantics for a constant is to create an object representing the constant. However, creating objects is memory intensive, so we perform a couple of optimizations for constants. Just before code generation, we perform a pass searching for constants. We insert code to generate the constant object at the beginning of the file, assign it to a uniquely named variable, and use that variable throughout the file.

```
(* Beginning of file *)
let _int_2 = pyint_new 2 in
  ...
_int_2
```

We also use an optimization implemented in CPython, creating and storing the first 100 integers when the runtime first executes.

**Function and method definition**  A function definition results in the creation of a function object which contains a piece of code. Every Python function code can be translated to an equivalent OCaml function with the same type signature.

```
obj \rightarrow obj \rightarrow obj
```

The first argument is a Python list object containing the positional arguments, while the second is a Python dictionary object containing the named keyword arguments. Every function returns an object, which may be the null object `None`.

As shown in Figure 5.2, we first insert code in the body of the generated function to process the function arguments. A common case is when a function is called with only positional arguments, whose number matches that of its formal parameters (handled by `assign\_pars`). In other cases, the arguments have to be reorganized to match with the correct formal parameters (handled by `process\_args`). We finally translate the Python function body into OCaml and place it in a function object which is added to the environment.

**Class definition**  A class declaration has three major components: the class name, its bases and the initial dictionary. The body of the class is first translated, inserting its attributes and methods into a new dictionary. The new class is then created using this dictionary and the base classes by a function `build\_class`.

There are three major steps in a new-style class definition:

- **Find the metaclass** The metaclass of a class determines the function to be used in its construction. By default, the metaclass is `type`, unless specified using the special variable `\_\_meta\_class\_\_`.

- **Find the primary base** For every class, one base class is determined as the primary base, from which all its default methods are inherited.
<table>
<thead>
<tr>
<th>Terms</th>
<th>Expression</th>
</tr>
</thead>
</table>
| **Constants** | \[ [c] = \begin{cases} 
  \text{pyint}_{\text{new}} c & \text{if } c \text{ is an int} \\
  \text{pyfloat}_{\text{new}} c & \text{if } c \text{ is a float} \\
  \text{pystring}_{\text{new}} c & \text{if } c \text{ is a string} \\
  \ldots 
\end{cases} 
|  |
| **Variables** | \[ [x] = \begin{cases} 
  \text{getattr } c \ x & \text{if } \text{scope} \text{ is class } c \\
  \text{global\_lookup} \ x & \text{otherwise} 
\end{cases} 
|  |
| \[ [e.x] = \text{getattr} [ [e] ] x \] |  |
| \[ [\text{global } x] = \text{global\_add} \ x \] |  |
| **Assignment** | \[ [l = c] = \begin{cases} 
  [l] := \text{ref} [c] & \text{if } \text{scope} \text{ is Function or Module} \\
  \text{setattr } c \ x & \text{if } \text{scope} \text{ is class } c 
\end{cases} 
|  |
| \[ [e_1.x = e_2] = \text{setattr} [ [e_1] ] x [e_2] \] |  |
| **Function** | \[ [\text{def}(\text{params}) : S] = \text{let } f\text{obj} = \text{create\_func} (\text{fun } pl \rightarrow \text{if } |pl| = |\text{params}| \land |kw| = 0 \text{ then } \text{assign\_pars} pl \text{ params} \text{ else } \text{process\_args} \text{ params} pl \text{ kw} ) ; \] \[ S \] \in \text{add\_local } f\text{obj} \] |  |
| **Class** | \[ [\text{class } c(\text{bases}) : S] = \text{let } c\text{obj} = \text{build\_class} [ "c", [bases], [S] ] \text{ in } \text{add\_local } c\text{obj} \] |  |
| **Loops** | \[ [\text{for } l \text{ in } e : S] = (iobj := \text{py\_iter} [e] ; \text{try} ( \text{while } \text{true} \text{ do } ( [l] := \text{py\_next} iobj ; \text{try} [S] \text{ with } \text{ContinueException} \rightarrow () \text{ done} ) \text{ with } \text{BreakException} \rightarrow () )); \] |  |
| \[ [\text{while } e : S] = \text{try} ( \text{while } [e] \text{ do } \text{try} [S] \text{ with } \text{ContinueException} \rightarrow () \text{ done} ) \text{ with } \text{BreakException} \rightarrow () ) \] |  |
| **Control flow** | \[ [\text{if } e_1 : S_1 \text{ elseif } e_2 : S_2 \text{ else } S_3] = \text{if } [e_1] \text{ then } [S_1] \text{ else } ( \text{if } [e_2] \text{ then } [S_2] \text{ else } [S_3] ) \] |  |
| [break] = \text{raise } \text{BreakException} \] |  |
| [continue] = \text{raise } \text{ContinueException} \] |  |
| [return ] = \text{raise} ( \text{ReturnException} [e] ) \] |  |
| [yield e] = (\text{event\_send} chan [e]; \text{event\_recv} chan) \] |  |
| [pass] = () \] |  |
| **Operators** | \[ [e_1 \ op \ e_2] = [\text{op}] [e_1] [e_2] \] |  |
| **Calls** | \[ [c_1 \text{ (pos, kwd)}] = \text{func\_call} [ [c_1] [e_1] [e_1] ] \] |  |

**Figure 5.** Translation semantics from source to target
• Initialize the methods The dictionary of the new type and its fields have to be consistent. In addition, the MRO (method resolution order) is computed to enable method lookups.

In CPython, the creation of each new type and its instance involves allocating a precise amount of memory and its initialization based on information in the metaclass and the base types. In our implementation, this process is simplified because all objects have the same size in our OCaml representation, and memory is automatically managed. We simply copy a prototype object (the primary base class) and modify the methods which have been changed. After the newly created type is initialized, we add it to the appropriate environment.

Loops Python provides the for and while iteration constructs. The fundamental object which controls any iteration in Python is the iterator object. This object can be generated by any sequence type, such as list or tuple, or any object which implements the __iter__ method. The for loop below iterates over the list l and prints its elements, assigning each in turn to i.

```python
for i in l:
  <body>

l_iter = iter(l)
while True:
  i = l_iter.next()
  <body>
```

This while loop is exactly equivalent to the for loop. Each invocation of the next() method of the iterator object l_iter returns the next element of its source sequence l. When there are no elements remaining, the next() method raises an internal StopIteration exception which terminates the loop normally. In addition, loops support non-local control flow constructs such as break, continue and return statements. We translate both while and for loops into while loops in OCaml, as shown in Figure 5.2.

OCaml does not provide break constructs or call/cc mechanisms for non-local control flow. In order to account for non-local control-flow constructs, we simply enclose the loop body with appropriate exception-handlers. We define two new OCaml exceptions; BreakException and ContinueException to represent break and continue statements respectively.

Generators A Generator function is the Python implementation of the functional programming concept of a lazy list. This is a potentially infinite list where each element is generated only once, on request, by a function that can be customized by the user.

```python
def pow2(N):
  for i in xrange(1,N):
    yield i^2

# print the first 10 powers of 2
for i in pow2(10):
  print i
```

A generator function has one or more yield statements in its body. When the generator is first called, it creates a generator object. Whenever this object is invoked using the next() function, it returns the argument of the subsequent yield and suspends execution. When the function returns normally, it raises a StopIteration exception signalling an end to the lazy list.

Implementing generator functions requires storing the entire state of a function as well as its execution pointer. The function should be able to pick up execution from that point and with that stored state when it is invoked again. Since yield statements can occur at any point in the function body, there could be several suspension and re-entry points for the function.

OCaml does not provide any direct facilities to implement generator functions. However, the semantics of generator functions can be easily implemented using threads. Threads provide the same facilities of suspension and state restoration that generators demand. However, performance using threads is highly implementation-dependent due to the overhead associated with them.

```python
[gfunc(args)] =
let chan = Channel.create() in
Thread.create (func_call [gfunc, [args]] ) chan
```

As shown above, a call to a generator function creates a new thread for the function and unique communication channel between the caller and the callee. This channel is used by the caller to invoke the callee, and the callee to return the next value. If any exceptions are raised by the function, they are also communicated through this channel. At the end of the function execution, the callee thread exits and the channel is destroyed.

5.3 Design of the compiler

We use a parser library from the CPython implementation to convert Python source to a standard string AST representation. After that, our implementation takes over and converts the string representation to an AST representation in OCaml. The ASTs are represented by tagged union data types in OCaml, with each tag representing a different kind of node.

![Figure 6. Design of the compiler](image)

Managing Scopes Python has three distinct scopes: Global, Class body and Function body. The Global scope includes that of the current file, but also imported modules and built-in functions in the runtime. The scope of each variable in Python can be determined statically. In our implementation, we first perform a pass over the AST to determine scopes. Each scope record (Figure 7) describes the variables and attributes which are defined, used, global or free in this scope. In addition, a scope contains information about its kind, its parent (enclosing) scope as well as a list of its children scopes. The rules for parent scopes are somewhat complex in the case of nested functions and classes.

During the scope analysis pass, we create a hash table of scope records, indexed by the AST node marking the beginning of the new scope. During the actual translation pass, the appropriate scope record can be easily extracted from this scope table.

Translation step The translation step traverses over the AST recursively, creating a new AST representation which is very close to the target language (OCaml). At each node, the current scope and any newly created scopes have to be taken into account. This stage
type scope =
  {
    mutable uses: Idset.t; (* set of names *)
    mutable defs: Idset.t;
    mutable params: Idset.t;
    mutable frees: Idset.t;
    mutable globals: Idset.t;
    mutable children : scope list;
    stype: scopetype; (* Module, class... *)
    parent: scope option;
  } (* Scope table *)
type scopes = (Ast.node, scope) Hashtbl.t

Figure 7. Representing scopes

translates operators and constructs in Python to equivalent functions which have been defined in the runtime environment (Figure 5.2). The final step is code generation, which creates an actual OCaml program.

6. Results

In this section, we compare the performance of our implementation against several other Python implementations on both Windows and Linux.

6.1 Python implementations

In addition to CPython and our OCaml implementation, we compare three other implementations in our evaluation: IronPython, PyPy and Jython. All these implementations are compatible with Python 2.2 or above. Our evaluation compares all these five platforms on Windows. On Linux, we compare CPython, OCaml and Jython.

IronPython IronPython is an implementation of Python written in C# for Microsoft’s .NET platform. It is based on Microsoft’s Common Language Runtime (CLR), which uses a typed low-level intermediate format. IronPython compiles a Python script to CLI, the common intermediate representation of CLR. The CLR compiler takes over and compiles the script down to native windows code. It is not portable across platforms and can be used only on Windows.

Jython Jython is an implementation of Python in the Java language (?). It aims to create a scripting language that is identical with Python at the syntax level, but can be seamlessly interfaced with Java. Currently, Jython is mostly compatible with Python 2.2, but some features are not fully implemented. It also does not support most standard Python libraries which are written in C, instead providing Java equivalents e.g. JNumeric for the CPython NumPy module.

PyPy PyPy is an implementation of Python in Python. The guiding idea is to translate a Python-level description of the Python language itself to lower level languages (?). The initial Python implementation is developed for a restricted subset of Python (RPython). A bootstrapping process is used to create a full Python implementation. PyPy aims to use a JIT compiler to translate Python to native code, but currently is a bytecode interpreter.

6.2 Benchmarks

We have used a diverse set of benchmarks in our comparative analysis, drawn from different benchmarking suites in the public domain.

Pybench This is a micro-benchmark suite which tests basic Python features such as integer and float operations, list subscripting and slicing and dictionary lookups. Each benchmark consists of a 50-100 simple operations run in a loop about 50k-100k times.

Pystone This is a standard Python benchmark testing classes, method and attribute lookups and assignment, and various other basic language features. It is 236 lines long.

Recursion We include the standard recursive factorial and fibonacci functions

Parrotbench This is a benchmark suite released by CPython developers to test competing implementations for both correctness and speed. It consists of seven benchmarks, out of which we currently run two completely.

6.3 Coverage

Table 1 shows the coverage of our benchmark suites by various Python implementations. Our OCaml-based implementation does not run all these benchmarks, mainly because of lack of coverage in the following areas

- **Built-in methods** Python has a large collection of built-in types, each providing many built-in methods. Currently, some built-in types such as `unicode` still remain to be implemented. Implementing these types and methods is a time-consuming process, but we aim to eventually support all of them. However, these remaining methods are independent of the features tested in the benchmarks which we do support. We do not expect our speedups to change significantly, though the increased size of the runtime may affect loading time slightly.

- **External libraries** Many Python programs rely on external libraries connected to Python via its foreign function interface. We fully intend to support these libraries eventually. It remains to be seen how they perform, since that depends on the design and implementation of our foreign function interface.

Among the current benchmarks, our implementation does not compile Parrotbench b0, b5 and b6 due since we have not implemented all the methods of the built-in type `string` and the `unicode` type. The remaining benchmarks that we do not compile depend on the first two. Jython does not run the parrotbench benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CPy</th>
<th>IronPy</th>
<th>PyPy</th>
<th>Jy</th>
<th>OCaml</th>
</tr>
</thead>
<tbody>
<tr>
<td>intops</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>floatops</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>dictops</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>listops</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>constructs</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>funcalls</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>instances</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>lookups</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>fact</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>fib</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>pystone</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>fannkuch</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>parrot b0</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>parrot b1</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>parrot b2</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>parrot b3</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>parrot b4</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>parrot b5</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>parrot b6</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
b1, b2 and b0 since its support for generator functions and some other newer Python features is not complete. PyPy fails with an error stating that some built-in methods are missing.

### 6.4 Discussion of results

Figures 8 and 9 give the speedups of the different implementations on Windows and Linux respectively, with CPython normalized to 1. We summarize some of our key observations.

- **Our OCaml implementation performs better than IronPython, PyPy and Jython over all benchmarks on both architectures.** Compared to CPython, we obtain speedups greater than 1 on most cases. The highest speedups (3-4) on both platforms are obtained on the benchmarks *intops*, which tests simple integer operations, and *constructs*, which tests basic *if, for* and *while* statements. On Linux, the only benchmark with a speedup less than 1 is *Pystone*. On Windows, OCaml performs worse than CPython on a few other benchmarks such as *tupleops* (speedup 0.67), which tests sequence slicing and *fib* (speedup 0.9), the fibonacci function. This may be a reflection of the underlying OCaml implementation, which performs better on the whole on Linux than Windows.

- **IronPython’s performance is uneven, ranging from speedup of almost 2 on *constructs* to as low as 0.03 on *jaanikuch*.** For many of the other benchmarks, the speedup is around 1, close to CPython. The IronPython website mentions that performance has been improving greatly over the last few years, mainly due to optimizations implemented by Microsoft at the intermediate language level of CLR.

- **On Windows, PyPy performs the poorest on the whole with the highest speedup of 0.55 on *listops*.** On most other benchmarks, the speedup of PyPy is around 0.3-less. One possible reason is that the version of PyPy currently available is a bytecode interpreter, since the native code compilers are still experimental. We expect that the performance of PyPy will improve once the its runtime environment, hurting performance. On Linux, the only benchmark with a speedup less than 1 is *Pystone*. On Windows, OCaml performs worse than CPython on a few other benchmarks such as *tupleops* (speedup 0.67), which tests sequence slicing and *fib* (speedup 0.9), the fibonacci function. This may be a reflection of the underlying OCaml implementation, which performs better on the whole on Linux than Windows.

- **Jython’s performance on Windows is better than PyPy, with a maximum speedup of 1.08 on *fact*.** For other benchmarks, the speedup is around 0.5. Jython has a noticeable overhead to load the its runtime environment, hurting performance. On Linux, Jython’s performance is much poorer than on Windows, with a highest speedup of 0.8 on *fact*. Most other benchmarks obtain speedups of only around 0.3. Since Jython is a java bytecode release, it is platform-independent. We may conclude that the performance gap between Windows and Linux is the result of the underlying VM implementation (Java 6 server on both platforms).

- **On the *Pystone* benchmark, which is accepted as an important standard in the Python community, CPython still performs the best.** Our implementation gets a speedup of 0.9 on both platforms. PyPy and Jython both obtain speedups of 0.2, while IronPython is about 0.5. Profiling our OCaml-generated executable using *gprof* reveals that a significant amount of time (over 10%) is spent in dictionary update operations, hence this is a possible bottleneck that we need to examine.

<table>
<thead>
<tr>
<th>Engine</th>
<th>OCaml</th>
<th>C (approx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parser+Compiler</td>
<td>2900</td>
<td>16000</td>
</tr>
<tr>
<td>Runtime</td>
<td>6800</td>
<td>50000</td>
</tr>
</tbody>
</table>

Table 2. Number of lines in OCaml vs C

**Code size** Table 2 shows the approximate size of our OCaml implementation compared to the equivalent implementation in CPython. We see that in case of both the translator and the runtime environment, the OCaml version is significantly smaller in size. This, combined with the fact the OCaml is a higher-level language than C, indicates that the productivity of a language developer using this methodology is higher.

### 6.5 Python-to-C translators

In addition to the Python implementations mentioned above, there are a few other projects which translate Python to native executable C/C++. The most popular ones among them are Shedskin, Psyco and the scipy.weave package. These projects each adopt different methods, however, all of them only accept programs which are statically typed. Shedskin uses a type inference engine to infer types of variables, thus only accepting programs which are ‘implicitly statically typed’. Psyco requires the programmer to insert C-like type declarations in Python programs. The weave package translates Python code literally to C, leaving it to the programmer to use the correct types. All of these techniques produce speedups of up to two orders of magnitude over their source Python code, however, the programmer has to forgo all the dynamic features of Python.

### 7. Related work

**Typed Intermediate Languages** The FOX project at CMU (?) and the FLINT project at Yale (?) aim at compiling any language to a typed intermediate language (TIL). Research on typed intermediate languages has focused on giving the programmer expressivity while preserving type safety. Specific examples of expressivity include ability to talk about registers, stack and memory locations of the underlying machine. TILs are meant as internal low-level
representations for compilers, providing a handle on the internal
details of the target hardware to achieve competitive performance.
The Singularity project at Microsoft Research (?) currently incor-
porates some of these ideas in its goal to build a safe operating
system.

Our motivation is similar in that we aim to provide a way to
achieve high performance for dynamic languages in a safe manner.
While much of the work in TILs aims at developing the language
and type system to ensure safety, our goal is to develop methods for
reusing these tools once they are sufficiently mature. While ML-
like languages have fairly limited type systems, they are high-level
languages with well-developed native code compilers and type in-
ference. This makes them an easier target language to translate to,
enabling compiler developers to focus on the most important opti-
mizations rather than myriad details. Notwithstanding these differ-
ences, our methods will be compatible with the use of TILs in place
of typed functional languages.

Other script compilation efforts There are several dynamic
scripting language compilation projects underway in the compiler
community.

Broadway The Broadway compiler project (?) at UT Austin
aims to provide compiler support for a wide range of domains
in the context of existing programming languages. This approach
uses library-level optimization to improve performance of scripting
languages, exploiting the domain-specific semantics of software
libraries. The key idea of Broadway is a separation of concerns:
compiler expertise is built into the Broadway compiler machinery,
while domain expertise resides in separate annotation files that are
provided by domain experts.

Telescoping Languages The Telescoping Languages project at
Rice University (?) aims to make high-performance programming
productive by constructing efficient domain-specific high-level lan-
guages from annotated component libraries. These languages are
called telescoping languages because they can be nested within one
another. The component libraries are exhaustively analyzed in ad-
vance to produce a language processor that recognizes and opti-
mizes library operations as primitives in the language. The key to
making this strategy practical is to keep compile times low by gen-
erating a custom compiler for each language with extensive built-in
knowledge of the underlying libraries.

These efforts are relevant to our work because compiler anno-
tations can be in the form of type signatures for specific functions.
A compiler for a static language such as OCaml can use these type
annotations at compile-time to improve performance of dynamic
language programs which have been translated to a statically typed
language.

8. Conclusions and Future work

In this paper, we propose that dynamic languages can be com-
plied to high-performance native code executables by translating
to typed functional languages and utilizing existing functional lan-
guage compilers. As proof-of-concept, we have built a compiler for
a popular dynamic language, Python, by translating it to OCaml, a
typed functional language.

Our OCaml compiler currently compiles many standard Python
benchmarks, however, there are several built-in methods remaining
to be implemented, along with the foreign function interface. Ta-
ble 3 summarizes our performance evaluation of several Python im-
plementations, comparing speedups against the standard CPython
implementation. Our OCaml compiler currently performs better
than CPython on most supported benchmarks, and better than the
other implementations on all benchmarks. However, as we add
more features and methods to the runtime, these numbers may
change. It must be noted that we do not currently perform any comp-
iler optimizations which may improve performance. The size of
our code base is five times smaller than the equivalent C code, indi-
cating higher programmer productivity. We have also provided a
formal representation of our source and target languages, and a
formal translation from the source to the target.

<table>
<thead>
<tr>
<th></th>
<th>IronPython</th>
<th>PyPy</th>
<th>Jython</th>
<th>OCaml</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best</td>
<td>1.92</td>
<td>0.55</td>
<td>1.08</td>
<td>3.92</td>
</tr>
<tr>
<td>Median</td>
<td>0.8</td>
<td>0.28</td>
<td>0.48</td>
<td>1.37</td>
</tr>
<tr>
<td>Worst</td>
<td>0.03</td>
<td>0.13</td>
<td>0.25</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Table 3. Speedup summary

As future work, our first priority is to provide complete sup-
port for all the built-in types and methods in standard Python, fol-
lowed by the foreign function interface for external libraries. Af-
ter that, we intend to investigate some compiler optimizations for
higher performance. This includes well-known optimization tech-
niques such as Tag elimination (?). Additionally, we would also
like to study whether methods such as gradual typing (?) can be
incorporated to benefit performance and programmer productivity.

10