UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GUSTAVO TAVARES CABRAL

Um otimizador estático para a linguagem Python

Trabalho de Conclusão apresentado como
requisito parcial para a obtenção do grau de
Bacharel em Ciência da Computação

Prof. Dr. Marcelo Johann
Orientador

Porto Alegre, 23 de dezembro de 2013

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Gustavo Tavares Cabral,

Um otimizador estático para a linguagem Python /

Gustavo Tavares Cabral. – Porto Alegre: Graduação em Ciência da Computação da UFRGS, 2013.

53 f.: il.

Trabalho de Conclusão – Universidade Federal do Rio Grande do Sul. Curso de Bacharelado em Ciência da Computação, Porto Alegre, BR–RS, 2013. Orientador: Marcelo Johann.

1. Python. 2. Otimização estática. 3. Compiladores. I. Johann, Marcelo. II. Título.

gtcabral@inf.ufrgs.br
http://inf.ufrgs.br/∼gtcabral

That which is static and repetitive is boring. That which is dynamic and random is confusing. In between lies art. John A. Locke

AGRADECIMENTOS

À minha família, pelo apoio e compreensão durante toda a minha vida.

Aos meus amigos, pelo companheirismo e por tornar a experiência universitária um pouco menos chata. Em especial a André Rodrigues Olivera e Francisco Gerdau de Borja, pela inestimável colaboração na realização deste trabalho.

Aos professores e servidores da UFRGS, pela dedicação e entusiasmo transmitidos diariamente.

À comunidade do *software* livre, por tornarem possível este trabalho. Python me deu um assunto, GNU Emacs um editor, LaTeX um documento bem formatado, GNU/Linux novas possibilidades, e OpenSSH uma maneira de fazer tudo isto de onde eu quiser. Sem eles, o que restaria?

Por fim, ao mundo e a todo mundo, fontes de inspiração, quase sempre de forma não intencional, para tudo o que fiz, faço e farei.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

SIGLAS

Ajax Asynchronous JavaScript and XML. 14
AOT Ahead-of-time. 25
API Application Programming Interface. 18
ASDL Abstract Syntax Description Language. 20, 35, 47
AST Abstract syntax tree. 8, 21, 33

DLL Dynamic Link Library. 24

GIL Global Interpreter Lock. 5, 20, 21, 37, 42
GMP GNU Multiple Precision Arithmetic Library. 41
GPU Graphics Processing Unit. 25, 27

JIT Just-in-time. 17, 25, 27, 28, 40, 42
JVM Java Virtual Machine. 17

PDF Portable Document Format. 14

RAII Resource Acquisition Is Initialization. 29

SSA Static Single Assignment. 42

RESUMO

Python é uma linguagem de programação dinâmica largamente utilizada nos mais variados domínios de aplicação, reconhecida por sua simplicidade e leve curva de aprendizado. Porém, esses pontos contrastam com o fato de que programas escritos na linguagem são, na maioria dos casos, muito mais lentos do que programas implementados em outras linguagens atuais.

Este trabalho tem o objetivo de solucionar parte desse problema, propondo um meio de otimizar trechos de código críticos para o desempenho. O processo consiste na conversão desses trechos para C++, uma linguagem estaticamente tipada que tem como característica a geração de programas eficientes.

Os resultados obtidos demonstram que o objetivo foi atingido, com uma melhora no desempenho de até 60 vezes nos testes realizados.

Palavras-chave: Python, otimização estática, compiladores.

An optimizing static compiler for the Python programming language

ABSTRACT

Python is a dynamic programming language widely used in several application domains, it is known by its simplicity and smooth learning curve. However these desirable features contrast with the fact that programs coded in python are, in most cases, much slower than programs implemented in other recent languages.

This work aims to solve part of this problem, proposing a way to optimize code snippets that are critical for performance. The process consists in translate these snippets to C++, a statically-typed language which has the characteristic of generating efficient programs.

The results show that the objective was achieved, with a performance boost of up to 60 times in the performed tests.

Keywords: Python, optimizing static compiler, compilers.

1 INTRODUÇÃO

Historicamente, linguagens dinâmicas construíram seu espaço com sucesso em tarefas limitadas por E/S, ao passo que tarefas limitadas por CPU geralmente são realizadas em linguagens estaticamente tipadas como C e C++.

Entretanto, esse panorama vem mudando. Com o aumento no uso de *frameworks* e Ajax, uma nova "guerra dos navegadores" foi iniciada, dessa vez pelo desempenho do motor Javascript. Os avanços alcançados permitiram que muitas aplicações, anteriormente disponíveis através de *plugins* escritos em linguagens estáticas, possam ser escritas diretamente em Javascript, como um renderizador de documentos PDF (PDF.js [1]), motor Flash (Shumway [2]) e inclusive decodificadores de vídeo (Broadway.js [3]).

Outro exemplo notável é a linguagem Julia [4]. Lançada em 2012, a linguagem tem o desempenho como uma das bases de projeto. Utilizando inferência de tipos em tempo de execução – juntamente com anotações de tipo opcionais – o desempenho alcançado é comparável ao de programas escritos em linguagens como C e Fortran.

1.1 Objetivos

Este trabalho visa criar um otimizador estático para a linguagem Python. Espera-se um ganho de desempenho médio de uma a duas ordens de magnitude em relação à interpretação pura.

Dentre as características desejadas para o projeto estão:

- Manutenção da sintaxe.
- Praticidade: as alterações de código necessárias para habilitar as otimizações devem ser aplicadas no próprio arquivo de código, dispensando o uso de ferramentas externas.
- Interoperabilidade entre os códigos otimizado e não otimizado.
- Facilidade para desabilitar as otimizações, para fins de depuração.
- Compatibilidade completa com a linguagem, mesmo em casos em que o otimizador não consegue transformar o código – permitindo que os recursos do otimizador sejam implementados de forma incremental, de acordo com a necessidade.

1.2 Estrutura do trabalho

Este trabalho é organizado da seguinte forma:

- no capítulo 2, exponho o contexto em que o trabalho está inserido, explanando as características essenciais para a elaboração do mesmo.
- no capítulo 3, apresento diversas alternativas com propostas similares.
- no capítulo 4, detalho a implementação do projeto, com destaque para a arquitetura e a especificação da linguagem, juntamente com as suas limitações.
- no capítulo 5, mostro os resultados obtidos nos testes realizados, comparando com as principais alternativas.
- no capítulo 6, realizo a conclusão do trabalho e listo algumas melhorias aplicáveis a este trabalho.

1.3 Arquivos complementares

Os arquivos que complementam este trabalho (código-fonte do otimizador e dos testes), bem como revisões deste documento, podem ser encontrados em `http://www.inf.ufrgs.br/~gtcabral/2013/tcc/`.

2 CONTEXTUALIZAÇÃO

Neste capítulo, realizo uma breve introdução à linguagem Python, e descrevo alguns elementos fundamentais para a execução deste trabalho.

2.1 A linguagem Python

Python é uma linguagem de programação de propósito geral, de alto nível e com tipagem dinâmica. A linguagem suporta múltiplos paradigmas de programação, com destaque para orientação a objetos, procedural e funcional. A linguagem vem sendo empregada em uma ampla variedade de domínios, e sua popularidade se deve a diversos fatores:

- Sintaxe clara e expressiva.
- Biblioteca padrão com "baterias inclusas"[1].
- Excelente documentação.
- Vasto ecossistema de bibliotecas e ferramentas.

A linguagem vem sendo amplamente utilizada pela comunidade científica. É raro encontrar bibliotecas para a área que não possuam *bindings* para Python (e muitas dessas bibliotecas são escritas na própria linguagem). Esse é um fato bastante curioso, visto que o mesmo não acontece com Ruby, uma linguagem com características similares.

Porém, Python dificilmente teria obtido tamanho sucesso se dependesse exclusivamente do desempenho do seu principal interpretador (CPython). Uma comparação do desempenho do CPython em relação a programas escritos em C++ (utilizando o compilador GNU g++) é mostrada na figura 2.1 [5]. Nota-se que a relação entre os tempos de execução ultrapassa uma ordem de magnitude na maioria dos testes.

[1] Termo utilizado pelos fãs da linguagem para caracterizar quão extensa é a sua biblioteca padrão.

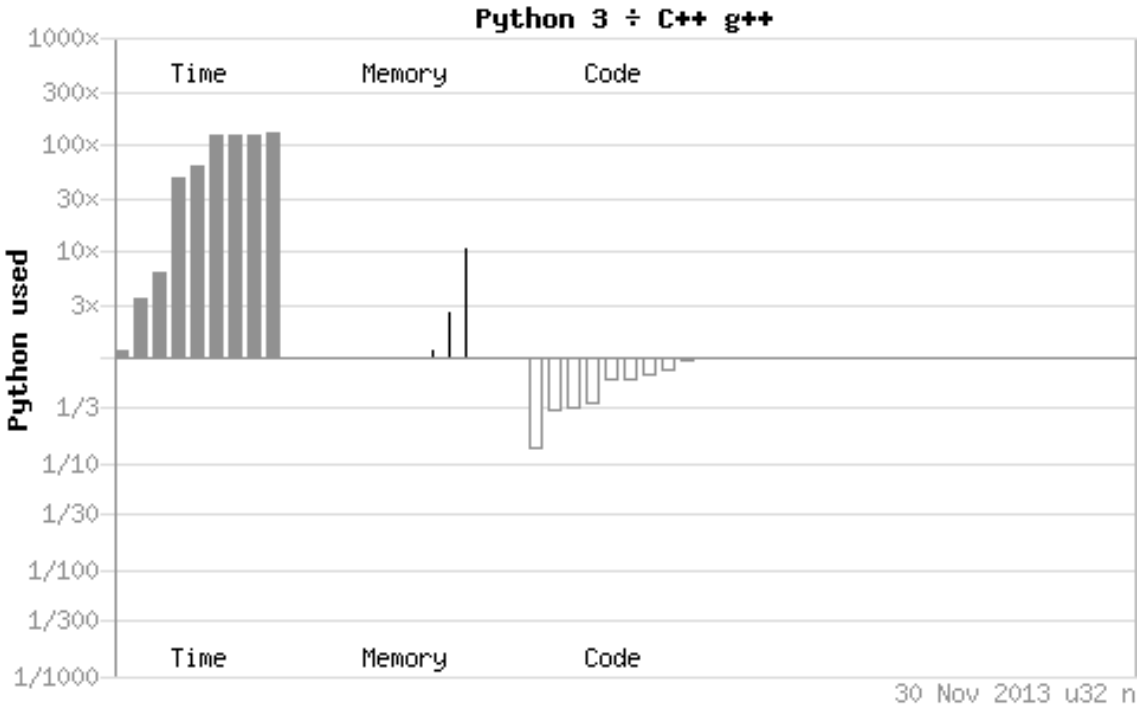

Figura 2.1: Comparação do desempenho entre códigos CPython 3 e C++ para um conjunto de *benchmarks*.

2.2 Principais implementações

A linguagem tem atualmente quatro implementações principais:

- CPython [6]: a implementação de referência, escrita em C, livre e de código aberto. É a implementação mais utilizada.
- Jython [7]: utiliza a máquina virtual Java (JVM), permitindo aos programas em Python total acesso a classes escritas em Java.
- IronPython [8]: utiliza o *Dynamic Language Runtime* [9], um *framework* para a implementação de linguagens dinâmicas para o .NET.
- PyPy [10]: uma implementação escrita em um subconjunto da própria linguagem (conhecido como RPython [11]), que pode ser compilado estaticamente. A implementação tem suporte a JIT.

Este trabalho dará foco a programas escritos para execução no CPython.

2.3 Versões

Em 2008, foi lançada a versão 3 da linguagem (conhecida como Python 3000, Python 3.0 ou Py3K). Essa versão quebrou fortemente a compatibilidade [12] com a linha de versões anterior (Python 2.x). Por essa razão, o desenvolvimento de ambas foi continuado paralelamente (a versão 2.7 foi lançada posteriormente, incluindo alguns recursos que visam facilitar a transição para as versões 3.x). A linha 2.x ainda é a mais utilizada e o padrão *de facto* na maioria das distribuições Linux/BSD [13], devido ao fato de muitas bibliotecas ainda serem incompatíveis com a linha 3.x.

Por ser mais organizada que a linha 2.x e por ser o “futuro” da linguagem, a linha 3.x

foi escolhida como base deste trabalho. A versão 3.3 foi adotada, por ser a mais recente.

2.4 Python/C API

CPython oferece uma rica API em C [14], permitindo tanto a possibilidade de embutir o interpretador em um programa escrito em C/C++ quanto a escrita de *extension modules* – módulos escrito em C ou C++ que utilizam a Python/C API para interagir com o interpretador e com código do usuário.

A Python/C API disponibiliza centenas de funções e macros que possibilitam a interação entre o código escrito em C/C++ e o interpretador. Boa parte delas tem como tipo dos argumentos ou de retorno `PyObject*`, um ponteiro opaco representando um objeto Python arbitrário.

A figura 2.2 exemplifica o uso da Python/C API, com a utilização do ponteiro opaco `PyObject*` e de duas funções da API (**PyArg_ParseTuple** e **PyLong_FromLong**). O retorno de um ponteiro nulo indica ao interpretador a ocorrência de uma exceção – **TypeError**, neste caso, disparada por **PyArg_ParseTuple**.

```
1  static PyObject*
2  spam_system(PyObject* self, PyObject* args) {
3      char const* command;
4      int sts;
5
6      if (not PyArg_ParseTuple(args, "s", &command))
7          return nullptr;
8      sts = system(command);
9      return PyLong_FromLong(sts);
10 }
```

Figura 2.2: Exemplo de função [15] demonstrando o uso da Python/C API.

```
1 >>> import spam
2 >>> status1 = spam.system('pwd')
3 /home/gustavo
4 >>> status2 = spam.system(234.5)
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in <module>
7 TypeError: must be str, not float
```

Figura 2.3: O uso de *extension modules* é transparente ao interpretador.

2.4.1 Gerência de memória

Para evitar vazamentos de memória, o CPython utiliza um contador de referências: todo objeto contém um contador, que é incrementado quando uma referência ao objeto é armazenada, e é decrementado quando uma referência é apagada. Quando o contador

chega a zero, a última referência ao objeto foi removida e a memória alocada para o objeto pode ser liberada.

Entretanto, esta abordagem não detecta ciclos (por exemplo, se a única referência a um objeto for possuída por ele mesmo, ele nunca seria liberado). Por isso, o CPython também dispõe de um detector de ciclos, que é executado periodicamente

O detector de ciclos de referência pode ser desabilitado em tempo de execução através do módulo ***gc*** da biblioteca padrão, útil caso o programador saiba que o seu código não possui ciclos de referência.

2.4.1.1 Manipulação do contador de referências

Em um código Python, o gerenciamento de memória é realizado automaticamente pelo interpretador. Entretanto, o mesmo não acontece nos códigos C/C++ que utilizam a Python/C API.

Para manusear o contador de referências, duas macros são definidas na Python/C API: **Py_INCREF** e **Py_DECREF**, que incrementam e decrementam o contador. **Py_DECREF** também libera a memória do objeto quando o contador chega a zero. Há ainda as macros **Py_XINCREF** e **Py_XDECREF**, que são seguras caso a variável passada como argumento seja um ponteiro nulo.

Em um retorno de função da Python/C API, duas possibilidades existem para o objeto retornado: o chamador recebe uma nova referência do objeto, ou recebe uma referência emprestada. No último caso, nada precisa ser feito ao contador de referências pelo chamador. A documentação da Python/C API é explícita em relação a qual tipo de referência é retornada por cada função.

De forma similar, em uma chamada de função também existem duas possibilidades: a função chamada pode tomar a referência dos objetos passados nos argumentos, ou pode não tomar. Quando a referência é tomada[2], a função chamada se torna responsável pela referência ao objeto, retirando este dever da função que chama.

Poucas funções da Python/C API tomam referências. Duas exceções notáveis são **PyList_SetItem()** e **PyTuple_SetItem()** (a lista completa pode ser encontrada no apêndice A.1). Estas funções foram projetadas desta forma porque é comum preencher listas e tuplas com objetos criados *in place*. Como exemplo, o trecho de código necessário para criar a tupla `(1, 2, 3, 'quatro')` pode ser escrito como mostra a figura 2.4.

```
1 PyObject* t;
2
3 t = PyTuple_New(4);
4 PyTuple_SetItem(t, 0, PyLong_FromLong(1L));
5 PyTuple_SetItem(t, 1, PyLong_FromLong(2L));
6 PyTuple_SetItem(t, 2, PyLong_FromLong(3L));
7 PyTuple_SetItem(t, 3, PyUnicode_FromString("quatro"));
```

Figura 2.4: **PyTuple_SetItem** toma a referência do objeto passado no terceiro argumento.

[2] A documentação oficial se refere a estes casos como *stealing a reference* [16].

Se estas funções não tomassem a referência dos objetos passados nos argumentos, o código deveria ser semelhante ao da figura 2.5.

```
1  PyObject *t, *var1, *var2, *var3, var4;
2
3  t = PyTuple_New(4);
4  var1 = PyLong_FromLong(1L);
5  var2 = PyLong_FromLong(2L);
6  var3 = PyLong_FromLong(3L);
7  var4 = PyUnicode_FromString("quatro");
8  PyTuple_SetItem(t, 0, var1);
9  PyTuple_SetItem(t, 1, var2);
10 PyTuple_SetItem(t, 2, var3);
11 PyTuple_SetItem(t, 3, var4);
12 Py_XDECREF(var4);
13 Py_XDECREF(var3);
14 Py_XDECREF(var2);
15 Py_XDECREF(var1);
```

Figura 2.5: Se **PyTuple_SetItem** não tomasse a referência, seria necessário decrementar posteriormente o contador de referências de cada objeto individualmente.

Embora a primeira alternativa seja mais interessante para quem escreve um módulo manualmente, a não uniformidade é prejudicial a um gerador automático de código. Na seção 4.2.1, descrevo como esta dificuldade foi contornada na implementação deste trabalho.

2.5 Geração da árvore de sintaxe abstrata

A biblioteca padrão de Python dispõe de um módulo (***ast***) para a geração da árvore de sintaxe abstrata a partir de uma *string* de código. A árvore gerada é baseada no código da ASDL [17] encontrado no apêndice A.2.

No módulo ***ast***, a hierarquia de classes da ASDL é representada utilizando classes em Python. A hierarquia de *BinOp*, por exemplo, é mostrada na figura 2.6.

Como a linguagem provê meios para acessar a árvore de sintaxe abstrata diretamente, não há necessidade de realizar as etapas prévias (como análise léxica e análise sintática). Considerando também as facilidades de introspecção da linguagem, conclui-se que é vantajoso que a alternativa proposta utilize a própria sintaxe da linguagem. A seção 4.4 mostra que esta intenção foi alcançada.

2.6 *Global Interpreter Lock*

Global Interpreter Lock [18] (GIL) é o mecanismo usado pelo CPython (e também por PyPy e Ruby MRI, a implementação de referência de Ruby) para impedir que duas ou mais *threads* executem o *bytecode* simultaneamente. Com isso, a implementação do

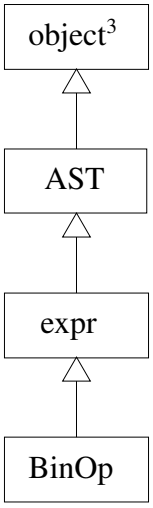

Figura 2.6: Hierarquia de classes para a classe *BinOp* da AST.

interpretador é simplificada, ao custo de restringir a possibilidade de paralelismo real em máquinas com múltiplos núcleos de processamento.

Tentativas de criar um interpretador livre do recurso não obtiveram sucesso, fato justificado pela perda de desempenho em programas que utilizam apenas uma *thread*. Entretanto, este é um tema recorrente nas discussões da comunidade, que determina como altamente desejadas em qualquer proposta de eliminação do GIL as seguintes características [19]:

- Simplicidade na implementação e manutenção do interpretador.
- Aumentar o desempenho de programas *multithread*, não reduzindo o desempenho de programas *single-thread*.
- Manter compatibilidade com a Python/C API.

Este trabalho não objetiva contornar o *Global Interpreter Lock*.

2.7 Detalhes sintáticos

Nesta seção, descrevo alguns elementos sintáticos da linguagem que serão utilizados na implementação (particularmente, na comparação de possibilidades existentes para a anotação dos tipos).

2.7.1 Comentários

Em Python, um comentário é iniciado por um caractere cerquilha (#) que não compõe uma *string*, e se estende até o fim da linha, como mostra a figura 2.7.

```
1 def square(x):
2     return x ** 2 # eleva x ao quadrado
```

Figura 2.7: Exemplo de código Python demonstrando o uso de comentários.

2.7.2 *Decorators*

Um *decorator* [20] é uma função (ou qualquer objeto que possua o método `__call__`) que retorna outra função. Tem como finalidade transformar funções ou classes. Um exemplo de uso é exposto na figura 2.8, no qual o *decorator* `sq_decorator` é aplicado à função `sq_inc1` através da sintaxe específica para o recurso (com `@`), e à função `sq_inc2` através de uma atribuição.

```
def sq_decorator(func):
    @functools.wraps(func)
    def inner_func(x):
        return func(x) ** 2
    return inner_func


@sq_decorator
def sq_inc1(x):
    return x + 1


def sq_inc2(x):
    return x + 1

# a sintaxe utilizando @ é um açúcar sintático
# para a seguinte transformação
sq_inc2 = sq_decorator(sq_inc2)
```

```
>>> sq_inc1(9)
100
>>> sq_inc2(9)
100
```

Figura 2.8: Descrição de uso de um *decorator*.

Decorators serão utilizados para anotar os tipos das variáveis locais nas funções a serem otimizadas (os detalhes são vistos na seção 4.4).

2.7.3 Anotações de função

Python 3 inseriu a funcionalidade de anotações de função [21], provendo uma maneira padronizada de anotar os parâmetros e o valor de retorno de uma função. A linguagem não define uma semântica associada às anotações, exceto que elas podem ser acessadas através do atributo `__annotations__` de sua função; são as bibliotecas – externas à biblioteca padrão – que dão semântica ao recurso, através de metaclasses e *decorators*. A figura 2.9 mostra a anotação do atributos `ham` e `eggs` (com os valores `42` e `int`), além da anotação do valor de retorno (a *string* `'Nothing to see'`).

```
>>> def f(ham: 42, eggs: int = 'spam') -> 'Nothing to see':
...     print('Annotations:', f.__annotations__)
...     print('Arguments:', ham, eggs)
...
>>> f('wonderful')
Annotations: {'eggs': <class 'int'>,
              'return': 'Nothing to see',
              'ham': 42}
Arguments: wonderful spam
```

Figura 2.9: Exemplo de código demonstrando o uso de anotações de função.

Anotações de função serão utilizadas na comparação das alternativas para anotação de tipos, vistas com detalhe na seção 4.4.

3 TRABALHOS RELACIONADOS

Neste capítulo, apresento brevemente algumas alternativas que também visam melhorar o desempenho de código Python.

3.1 Alternativas

Uma otimização básica que pode ser realizada é aplicada diretamente sobre a árvore de sintaxe abstrata, realizada pela seguinte biblioteca:

- ***astoptimizer*** [22]: Otimizador estático para a árvore de sintaxe abstrata gerada pelo interpretador. Opera sobre expressões como `1 + 2 * 3` e `len('abc')`, transformando em expressões que podem ser avaliadas mais rapidamente (nestes casos, os literais `7` e `3`).

Outra possibilidade é a escrita de código em outra linguagem, cuja integração com o interpretador é realizada por ferramentas que facilitam a interoperabilidade destas linguagens com Python:

- ***ctypes*** [23] e ***CFFI*** [24] (*Common Foreign Function Interface*): Permitem a integração de *shared libraries* (DLLs) com o código Python.
- ***SWIG*** [25] e ***f2py*** [26]: Geradores de interfaces C++ e Fortran.
- ***Boost.Python*** [27]: Biblioteca para C++ que facilita a interoperabilidade entre C++ e Python.
- ***Babel*** [28]: Ferramenta de interoperabilidade entre linguagens, de alto desempenho. Atualmente, suporta C, C++, Fortran 77, Fortran 90/95, Fortran 2003/2008, Java e Python.
- ***SIP*** [29]: Gerador de *extension modules* similar ao ***SWIG***, mas específico para Python.
- ***Pyfort*** [30]: Ferramenta que auxilia a criação de extensões usando rotinas em Fortran.

Um paradigma similar ao anterior é a inserção *inline* de código escrito em outras linguagens:

- ***Instant*** [31]: Permite *inlining* de código C e C++ em Python. Criado para o *FEniCS Project* [32], uma coleção de ferramentas focada na solução automatizada de equações diferenciais através do método dos elementos finitos – problema tipicamente de alto custo computacional.
- ***PyInline*** [33]: Como o ***Instant***, permite *inlining* de código de outras linguagens

diretamente no código Python.
- ***weave*** [34]: Módulo incluído no pacote ***SciPy*** [35] que provê ferramentas para utilizar código C/C++ em Python. Além de inserção *inline*, oferece como funcionalidades a tradução de expressões da biblioteca ***NumPy*** [36] para C++ e facilita a criação de *extension modules*.

Uma abordagem interessante é encontrada nos compiladores que suportam apenas um subconjunto da linguagem. Nestes, o código gerado é independente do interpretador Python:

- ***Pythran*** [37] e ***Shed Skin*** [38]: Compiladores Python para C++. Embora não necessitem de alteração no código original, ambas são restritas a um subconjunto da linguagem, tanto em relação à sintaxe/semântica quanto em relação às bibliotecas disponíveis.
- ***unPython*** [39]: Compilador de um subconjunto de Python (com anotação de tipos) para C.

O significativo uso da linguagem pela comunidade científica motivou a criação de bibliotecas específicas:

- ***Blaze*** [40]: Promete ser "a nova geração" do ***NumPy***. Dentre outros recursos, contém um compilador para uma linguagem própria, estaticamente tipada (baseada em Python e C).
- ***Parakeet*** [41]: Compilador em tempo de execução para computação científica (sobre ***NumPy***, basicamente). Bastante similar ao projeto ***Numba***, que será detalhado na subseção 3.1.2.
- ***Copperhead*** [42]: Projeto da NVIDIA Research que objetiva facilitar a paralelização de código em Python. Suporta atualmente GPUs da NVIDIA, tão como CPUs *multicore* (através das bibliotecas OpenMP[43] e Threading Building Blocks[44]).
- ***Theano*** [45]: Permite definir, otimizar e avaliar expressões matemáticas envolvendo *arrays* multidimensionais de maneira eficiente.
- ***minivect*** [46]: Mini compilador para expressões vetoriais (expressões sobre *arrays* multidimensionais nos quais a expressão é aplicada individualmente a cada item do *array*). Utiliza C e LLVM IR como representações intermediárias.
- ***numexpr*** [47]: Similar ao ***minivect***, mas utiliza uma máquina virtual própria (com suporte a JIT e a *multi-threading*). Também suporta a *Intel Vector Math Library* [48].

Na direção oposta, algumas soluções buscam uma melhora geral no desempenho, através de técnicas como compilação JIT e AOT.

- ***GCC Python Front-End*** [49]: Projeto recente que visa criar um compilador *ahead-of-time* para Python.
- ***Psyco*** [50]: Compilador JIT bastante popular na última década. O projeto foi descontinuado (o *site* oficial recomenda o uso do ***PyPy***).
- ***Unladen Swallow*** [51]: Baseado no código-fonte do CPython 2.6.1, tinha o objetivo de ser um compilador JIT para a linguagem (através do uso da LLVM). Apesar de ser patrocinado pelo Google, o projeto foi descontinuado.

Alinhados com o objetivo deste trabalho aparecem alguns compiladores estáticos para Python:

- ***2c-python*** [52]: Compilador estático de Python para C. Não utiliza declaração explícita dos tipos pelo programador; em vez disto, considera todas as variáveis como

`PyObject*`.

- ***Nuitka*** [53]: Similar ao projeto ***2c-python***, compila para C++ todas as construções que o CPython oferece (acessando o interpretador durante a execução). Há planos para inserir, nas próximas versões, inferência automática de tipos.
- ***mypy*** [54]: Linguagem de programação baseada em Python que combina tipagem dinâmica e estática.

Devido à maturidade e a semelhanças com este trabalho, as bibliotecas ***Cython*** e ***Numba*** serão descritas mais detalhadamente nas próximas subseções.

3.1.1 Cython

Cython é um compilador estático para Python, que objetiva facilitar a criação de extensões em C através de uma linguagem com sintaxe bastante similar à de Python.

Dentre as suas características, algumas motivaram a criação deste trabalho:

- interoperabilidade entre os códigos Python e Cython.
- ganho expressivo de desempenho através do uso de declarações estáticas de tipos.

Boa parte das vantagens que podem ser alcançadas com a biblioteca se baseiam no uso de sua linguagem específica [55], com sintaxe similar à da linguagem C – sendo, portanto, incompatível com Python. Um exemplo do uso da biblioteca é exibido na figura 3.1.

```
1  cdef double f(double x):
2      return x**2-x
3
4  cpdef double integrate_f(double a, double b, int N):
5      cdef int i
6      cdef double s, dx
7      s = 0
8      dx = (b-a)/N
9      for i in range(N):
10         s += f(a+i*dx)
11     return s * dx
```

Figura 3.1: Trecho de código Cython.

Entretanto, essa abordagem apresenta problemas que inviabilizam o uso do otimizador em larga escala:

- A biblioteca é mantida por uma comunidade pequena; defeitos que resultam em um comportamento inesperado têm uma probabilidade maior de ocorrer nestes ambientes (comparando com o CPython, por exemplo), deixando o programador "preso" à biblioteca e sua linguagem específica.
- Detalhes na sintaxe de Python são alterados com frequência, tornando necessárias eventuais mudanças na biblioteca .
- Dificuldade para depuração (mescla códigos C, Python e Cython).

Tais adversidades podem ser sanadas por uma alternativa que:

- Não altere a sintaxe da linguagem.

- Possa ser desabilitada facilmente, com pouca alteração no código-fonte – de preferência, uma solução em uma linha.

3.1.2 Numba

Numba é um compilador JIT para LLVM de código Python anotado (através de *decorators*). A biblioteca realiza especialização de tipos a partir dos argumentos de entrada – não há necessidade de anotação estática de tipos. A figura 3.2 mostra um exemplo de aplicação da biblioteca, com o uso do *decorator* `autojit`.

```
1  from numba import autojit
2
3  @autojit
4  def sum2d(arr):
5      M, N = arr.shape
6      result = 0.0
7      for i in range(M):
8          for j in range(N):
9              result += arr[i,j]
10     return result
```

Figura 3.2: Exemplo de código ***Numba***.

A biblioteca também permite definição estática de tipos pelo uso de anotações de tipo, o que é exemplificado na figura 3.3 com a passagem do argumento `locals` para `autojit`.

```
1  import numba
2
3  @numba.autojit(locals=dict(arr=numba.double[:,:]))
4  def sum2d(arr):
5      M, N = arr.shape
6      result = 0.0
7      for i in range(M):
8          for j in range(N):
9              result += arr[i,j]
10     return result
```

Figura 3.3: ***Numba*** também permite anotação estática de tipos.

Uma das principais finalidades do projeto é otimizar código que utiliza a biblioteca ***NumPy***.

Numba também possui uma versão paga (***NumbaPro*** [56]), que oferece a possibilidade de paralelização em arquiteturas *multicore* e GPUs.

Este trabalho tem objetivos parecidos, embora mais modestos do que os de ***Numba***. Enquanto ambos compartilham a ideia da manutenção da sintaxe de Python, a biblioteca

Numba realiza inferência automática de tipos e compilação JIT, propósitos não ambicionados para este trabalho.

Por outro lado, ***Numba*** não implementa um meio fácil de desabilitar a sua funcionalidade. Considerando o fato de que é uma biblioteca pouco madura (como pode ser visto no capítulo 5, ***Numba*** não foi capaz de executar nenhum dos três testes propostos), é um recurso valioso e simples de ser implementado.

4 IMPLEMENTAÇÃO

Neste capítulo, relato os detalhes da implementação do otimizador, juntamente com as suas limitações.

4.1 Linguagem intermediária

Para linguagem intermediária, a opção escolhida foi gerar código em C++. A escolha por C/C++ se deve ao fato de Python prover meios para a criação de extensões em C (através da Python/C API). C++ é preferível em relação à linguagem C pela possibilidade de personalizar os construtores/destrutores, recurso utilizado para facilitar o gerenciamento de memória (contagem de referências).

4.2 Tipos

4.2.1 Tipos intermediários

Para simplificar o gerenciamento de memória, todas as chamadas de funções da Python/C API que retornam um valor do tipo `PyObject*` são realizadas indiretamente, através de um tipo intermediário. Este passo realiza a conversão dos objetos retornados por cada função para um dos dois tipos criados para este fim:

- **MyPyObject_NewReference**: Utilizado nas funções que retornam uma nova referência.
- **MyPyObject_BorrowedReference**: Utilizado nas funções que retornam uma referência "emprestada".

Além destas transformações, funções que tomam a referência dos objetos passados como argumento também são modificadas, utilizando os seguintes tipos:

- **MyPyObject_StealReference**: Utilizado nos argumentos cuja referência do objeto passado é tomada pela função chamada.
- **MyPyObject_DontStealReference**: Utilizado nos outros argumentos, sendo atualmente apenas um *alias* para `PyObject*`.

Esses quatro tipos realizam o ajuste necessário ao contador de referências no momento em que seus objetos são criados ou destruídos. Tal padrão de projeto, conhecido como *Resource Acquisition Is Initialization* (RAII), é utilizado de forma similar nos *Smart Pointers* [57, 20.7: *Smart pointers*] e na classe `std::lock_guard` [57, 30.4.2.1: *Class*

template lock_guard]) de C++11.

A fim de realizar a indireção de tipos necessária, quatro alternativas foram consideradas.

4.2.1.1 Alternativa #1

A solução mais imediata é criar uma nova função para cada função existente na Python/C API, realizando as conversões necessárias nos tipos dos argumentos e no tipo de retorno. A figura 4.1 mostra um exemplo desta proposta, com a adaptação tanto para os tipos dos argumentos de entrada (**MyPyObject_StealReference** e **MyPyObject_DontStealReference**) quanto para o tipo de retorno (no caso, **MyPyObject_NewReference**).

```
1 MyPyObject_NewReference
2 My_foo(MyPyObject_StealReference a,
3         MyPyObject_DontStealReference b) {
4     return foo(a, b);
5 }
```

Figura 4.1: Uma nova função é definida para cada função da Python/C API.

Esta alternativa é suficiente quando o número de argumentos é previamente conhecido. Entretanto, algumas funções da Python/C API requerem um número variável de argumentos, como o exemplo exibido na figura 4.2.

```
1 PyObject* Py_BuildValue(char const* format, ...);
```

Figura 4.2: Exemplo de função da Python/C API com um número arbitrário de argumentos.

4.2.1.2 Alternativa #2

A Python/C API define funções equivalentes para algumas funções da Python/C API que recebem um número arbitrário de argumentos, sendo diferenciadas por receber um argumento do tipo **va_list** (tipo da linguagem C que armazena os argumentos passados a uma função com um número arbitrário de argumentos). Para estes casos, outra possibilidade é exibida na figura 4.3.

```
1 MyPyObject_NewReference
2 MyPy_BuildValue(char const* format, ...) {
3     va_list args;
4     va_start(args, format);
5     MyPyObject_NewReference const res =
6         Py_VaBuildValue(format, args);
7     va_end(args);
8     return res;
9 }
```

Figura 4.3: Exemplo de função que utiliza **va_list**.

4.2.1.3 *Alternativa #3*

Recurso introduzido no C++11, o uso de *variadic templates* [57, 14.5.3: *Variadic templates*] (*templates* que aceitam um número arbitrário de argumentos) é elegante e seguro, visto que a verificação dos argumentos é realizada em tempo de compilação. Um exemplo pode ser visto na figura 4.4.

```
1 template <typename... T>
2 MyPyObject_NewReference
3 MyPy_BuildValue(char const* format, T... args) {
4     return Py_BuildValue(format, args...);
5 }
```

Figura 4.4: Exemplo de função utilizando *templates* variádicos de C++11.

O uso de *templates*, entretanto, habitualmente aumenta o tempo de compilação.

4.2.1.4 *Alternativa #4*

Uma possibilidade mais simples e que não torna a compilação mais lenta é utilizar a macro **__VA_ARGS__**, definida em C99 e C++11. Seu uso é exemplificado na figura 4.5.

```
1 #define MyPy_BuildValue(...) \
2     MyPyObject_NewReference(Py_BuildValue(__VA_ARGS__))
```

Figura 4.5: Exemplo utilizando **__VA_ARGS__**.

Pela simplicidade e por suportar um número de argumentos arbitrário, este foi o método escolhido para realizar a indireção de tipos. Todavia, esta abordagem não trata funções da Python/C API que tomam a referência dos objetos passados como argumento. Como nenhuma delas aceita um número arbitrário de argumentos, tais funções são tratadas individualmente pela primeira alternativa citada.

4.2.2 Tipos para a geração de código

4.2.2.1 *Tipo para as variáveis locais*

Graças aos quatro tipos intermediários, a geração de código é simplificada, sendo necessária a criação e utilização de apenas um tipo (**MyPyObject_Local**), que substitui o uso de **PyObject***. O tipo criado tem os seguintes requisitos:

- Possuir operadores – como construção, atribuição e comparação - que aceitem objetos de **MyPyObject_NewReference** e **MyPyObject_BorrowedReference** (retornados pelas funções da Python/C API), além de objetos do próprio tipo **MyPyObject_Local**.
- Admitir ser transformado nos tipos dos argumentos utilizados nas funções da Python/C API (**MyPyObject_StealReference** e **MyPyObject_DontStealReference**).

A realização destes requisitos é facilitada com o uso de classes em C++, através da personalização de construtores e destrutores e sobrecarga de operadores.

4.2.2.2 *Tipo de retorno*

Ao retornar um objeto, é necessário que o contador de referências não seja decrementado. Para isto, o tipo **MyPyObject_Return** foi criado. Ele é utilizado apenas como tipo de retorno e não deve ser utilizado na declaração de variáveis locais. A figura 4.6 ilustra um modelo básico do código gerado pelo otimizador, uso dos dois tipos utilizados para geração de código.

```
1  inline MyPyObject_Return // tipo de retorno
2  spam_system_aux(PyObject* self, PyObject* args) {
3      char const* command;
4      int sts;
5      MyPyObject_Local res; // tipo das variáveis locais
6
7      if (MyPyArg_ParseTuple(args, "s", &command) == nullptr)
8          return nullptr;
9      sts = system(command);
10
11     // funções da C API não são chamadas diretamente
12     res = MyPyLong_FromLong(sts);
13     return res;
14 }
15
16 static PyObject*
17 spam_system(PyObject* self, PyObject* args) {
18     // conversão implícita de MyPyObject_Return
19     // para PyObject* no retorno
20     return spam_system_aux(self, args);
21 }
```

Figura 4.6: Modelo de código gerado pelo otimizador.

4.3 *Name mangling*

Desde a versão 3.0, Python suporta o uso de caracteres não-ASCII em identificadores [58]. Embora o padrão C++11 [57] também permita o uso de caracteres *unicode* [59] em identificadores, este suporte não está implementado nas versões atuais dos compiladores g++ e clang[1].

Por esta razão, e também para não haver conflito com palavras-chave e identificadores de C++, é necessário um mecanismo que realize um mapeamento entre o nome dos identificadores de Python e C++.

O mecanismo criado é composto por três regras. As regras são aplicadas separadamente a cada caractere *unicode*:

1. Para caracteres em 0-9, A-Z, a-z: Nenhuma alteração é realizada.
2. Para o caractere _: O caractere é duplicado.

[1] g++ 4.8.1 e clang 3.3

3. Para os outros caracteres: Ocorre a substituição por **_Uxxxxxxxx**, sendo **xxxxxxxx** o *codepoint* do caractere em hexadecimal.

Além destas regras, todos os nomes ganham o prefixo **pyname_**, para impedir que nomes como **void**, reservados em C++ mas permitidos em Python, sejam utilizados.

4.4 Interface com o usuário

Para realizar a interface com o usuário, diversas alternativas foram avaliadas.

A primeira delas, similar ao que é feito na biblioteca ***pythran***, usa comentários precedendo a definição de cada função a ser compilada estaticamente, como pode ser visto na figura 4.7.

```
1 #pythran export matrixmult(float list list, float list list)
2 def matrixmult(m0, m1):
3     ...
```

Figura 4.7: Alternativa de anotação de tipos utilizando comentários.

Embora concorde com as características essenciais do projeto, o uso de comentários apresenta algumas inconveniências:

- É tolerante a erros de sintaxe dentro dos comentários.
- Comentários são descartados pelo módulo ***ast*** durante a geração da árvore de sintaxe abstrata; seria necessária uma nova etapa de *parsing*.
- Não permite definir os tipos das variáveis locais.
- A correspondência entre um argumento e seu respectivo tipo é definida apenas pela ordem declarada, o que dificulta a leitura e é suscetível a erros caso a lista de argumentos da função seja alterada.

Outra possibilidade, utilizada pela biblioteca ***Numba***, é a passagem de uma *string* com a declaração dos tipos através de um *decorator*, como mostra a figura 4.8.

```
1 from numba import jit
2
3 @jit('f8(f8[:])')
4 def sum1d(my_double_array):
5     ...
```

Figura 4.8: Exemplo de anotação de tipos utilizando *strings*.

Este método tem a vantagem de ser totalmente suportado pela AST, além de não ser necessário analisar todo o arquivo (a chamada do *decorator* é realizada logo após a definição da função a qual ele se aplica). Entretanto, também manifesta alguns problemas:

- Erros de sintaxe na *string* passada como argumento devem ser tratados manualmente.
- A sintaxe é obscura.

- Como na alternativa anterior, não permite definir o tipo das variáveis locais, e a correspondência entre um argumento e seu tipo é dada apenas pela ordem declarada.

O módulo ***Numba*** suporta também o uso da mesma sintaxe sem o uso de uma *string*. A figura 4.9 apresenta um exemplo.

```
1 from numba import f8, jit
2
3 @jit(f8(f8[:]))
4 def sum(arr):
5     ...
```

Figura 4.9: Exemplo de anotação de tipos similar ao da figura 4.8, mas sem o uso de *strings*.

Agora, os erros de sintaxe são tratados diretamente pelo interpretador Python. Todavia, apresenta todos os problemas do método anterior, além de poluir o *namespace* global com os nomes dos tipos que serão usados (no exemplo, o tipo `f8`).

Uma alternativa bastante diferente das anteriores é uso de anotações de função. Uma das características interessantes do método é a possibilidade de ser aplicado parcialmente, como mostra a figura 4.10.

```
1 # Os argumentos 'spam' e 'eggs' não foram anotados
2 def my_pow(b: float, e: int, spam, eggs) -> float:
3     ...
```

Figura 4.10: Alternativa de tipagem utilizando *annotations*.

Contudo, é uma escolha inviável por não poder ser aplicado às variáveis locais.

A biblioteca ***Cython*** provê, além do modo de uso "clássico", um modo de uso em Python puro (com *decorators*), como demonstra a figura 4.11.

```
1 import cython
2
3 @cython.returns(cython.int)
4 @cython.locals(a=cython.double,
5                b=cython.double,
6                n=cython.p_double)
7 def foo(a, b, x, y):
8     ...
```

Figura 4.11: Alternativa de anotação de tipos utilizando dois *decorators* aninhados.

Este sistema tem inúmeras vantagens em relação aos anteriores: é bastante claro, não é suscetível a erros de sintaxe e permite definir apenas os tipos das variáveis de interesse (nota-se a ausência de definição para `x` e `y`, além de permitir a anotação para a variável local `n`).

A biblioteca ***Numba*** também permite o uso de uma sintaxe semelhante ao modo puro de ***Cython***. A figura 4.12 exibe um exemplo.

```
1 from numba import double, autojit
2
3 @autojit(locals=dict(mydouble=double))
4 def call_method(obj):
5     ...
```

Figura 4.12: Anotação de tipos utilizando apenas um *decorator*.

A alternativa escolhida é bastante similar a esta última, tendo as seguintes características:

- Suporte total do interpretador.
- A sintaxe é clara – apesar de verbosa – e reduz a possibilidade de erros cometidos pelo programador.
- Pode ser expandida, com o intuito de utilizar outros recursos além da anotação de tipos (habilitar *bounds checking*, por exemplo).
- Segue um popular princípio de projeto da linguagem: *"There should be one– and preferably only one –obvious way to do it."* [60].

Um exemplo do método selecionado para a implementação é mostrado na figura 4.13.

```
1  @pyoptimizer.optimize(
2      locals=dict(mydouble=compilertypes.float64,
3                  myfloat=compilertypes.float32,
4                  i=compilertypes.int32),
5      return_type=compilertypes.int64,
6      noexcept=True,  # a sintaxe é expansível
7                      # para recursos futuros
8  )
9  def func1(mydouble, myfloat):
10     ...
```

Figura 4.13: Método de anotação de tipos escolhido para ser utilizado neste trabalho.

4.5 Arquitetura

Para cada tipo especificado na subseção A.3.1, foram criadas classes Python para todas as subclasses de *expr* da ASDL[2]. Desta forma, o processamento necessário para as subexpressões de cada tipo é implementado em métodos de sua respectiva classe.

[2] As subclasses de *expr* são *Attribute*, *BinOp*, *BoolOp*, *Bytes*, *Call*, *Compare*, *Dict*, *DictComp*, *Ellipsis*, *GeneratorExp*, *IfExp*, *Lambda*, *List*, *ListComp*, *Name*, *Num*, *Set*, *SetComp*, *Starred*, *Str*, *Subscript*, *Tuple*, *UnaryOp*, *Yield* e *YieldFrom*.

4.5.1 Estágios

A transformação de código é realizada em estágios. Cada estágio recebe como entrada uma árvore de sintaxe abstrata e uma variável de contexto (um dicionário contendo, entre outras informações, o tipo informado das variáveis locais). Cada estágio retorna uma árvore de sintaxe abstrata e um dicionário de contexto, ambos possivelmente transformados. As transformações realizadas em cada estágio são descritas a seguir:

4.5.1.1 Estágio 1

- A toda expressão é atribuído um tipo.
- As expressões são convertidas para a Python/C API. Por exemplo, se a variável `num` não possuir um tipo anotado, a expressão `num + 3` é transformada em **MyPyNumber_Add(pyname_num, 3)**.
- As expressões são de-planificadas. Isto acontece com expressões do tipo *BoolOp* (`a1 and a2 and a3`) e *Compare* (`1 < 2 == 2 <= 3`). Para o caso do *Compare*, há a necessidade de utilizar variáveis temporárias, como mostra o exemplo da figura 4.14.

```
def func_compare():
    if f() < g() < h():
        ...
```

```
void func_compare_bad() {
    if (f() < g() and
        g() < h()) { // a função g é chamada duas vezes;
                     // comportamento difere do esperado
        ...
    }
}

void func_compare_good() {
    auto const temp1 = f();
    auto const temp2 = g();
    if (temp1 < temp2) {
        auto const temp3 = h();
        if (temp2 < temp3) {
            ...
        }
    }
}
```

Figura 4.14: Exemplo de de-planificação de um *Compare*, demonstrando a necessidade de utilizar variáveis temporárias.

- Algumas sentenças – como atribuições e laços – são transformadas. Por exemplo: o trecho `seq[x] = 1234` é transformado em **MyPyObject_SetItem(seq, x, 1234)**; já laços **`for`** que iteram sobre objetos cujo tipo não foi anotado são

substituídos por `while`, utilizando ainda funções como **MyPyObject_GetIter** e **MyPyIter_Next**.

4.5.1.2 Estágio 2

No segundo estágio, os tipos dos argumentos contidos nas chamadas à Python/C API são corrigidos. Por exemplo: ocorre a transformação de **MyPyNumber_Add(pyname_num, 3)** para **MyPyNumber_Add(pyname_num, MyPyLong_FromLong(3))**.

4.5.1.3 Estágio 3

Neste estágio, ocorre a geração do código C++.

4.6 Especificação da linguagem

O apêndice A.3 enumera todos os elementos da linguagem suportados pelo otimizador.

4.7 Limitações

Este trabalho não tem como objetivo oferecer paralelismo real ao CPython (limitado pelo *Global Interpreter Lock*, conforme detalhado na seção 2.6. Entretanto, nada impede o uso de módulos como ***multiprocessing*** e ***concurrent.futures*** (disponíveis na biblioteca padrão), que alcançam paralelismo real através do emprego de subprocessos.

A maior parte dos recursos da linguagem não é suportada. Segue uma listagem não exaustiva:

- Descritores
- Exceções
- *Generators* [61, 62]
- *Generator Expressions* [63]
- Gerenciadores de contexto [64]
- Funções e classes aninhadas
- Lambdas
- *List [65]/Set/Dict [66] comprehensions*
- Metaclasses [67]
- Metaprogramação

4.8 Revisão das características desejadas

O capítulo 1 lista cinco características fundamentais do projeto. Esta seção descreve brevemente como cada ponto foi atingido.

4.8.1 Manutenção da sintaxe

Todas as alternativas propostas para a anotação de tipos – discutidas na seção 4.4 – cumprem esta característica.

4.8.2 Praticidade

Para realizar a otimização de uma função, apenas o arquivo que contém tal função deve ser modificado. Não há uso de ferramentas externas.

4.8.3 Interoperabilidade entre os códigos otimizado e não otimizado

Esta propriedade é assegurada pela riqueza da Python/C API, que disponibiliza as funcionalidades do interpretador ao código C/C++.

4.8.4 Facilidade para desabilitar o otimizador

A implementação desta característica remete ao sistema de importação de módulos [68] utilizado pelo Python. A cada importação, uma *cache* de módulos previamente importados é verificada; se o módulo desejado já foi importado, uma referência ao módulo é retornada. Desta forma, existe um estado global compartilhado entre os diversos arquivos importados direta ou indiretamente pelo arquivo principal. Sendo assim, é suficiente realizar a chamada para desabilitar o otimizador (a função **disable**) apenas no arquivo principal, antes de qualquer outra importação, como mostra a figura 4.15.

```
1 import pyoptimizer
2 pyoptimizer.disable()
3
4 ...
```

Figura 4.15: Trecho de código demonstrando como desabilitar o otimizador.

Após a desabilitação, os *decorators* apenas passam adiante os argumentos recebidos. Tal comportamento é exibido na figura 4.16.

```
1 def bypass_decorator(func):
2     return func
3
4 @bypass_decorator
5 def square(x):
6     return x ** 2
```

```
1 >>> square(10)
2 100
```

Figura 4.16: Exemplo de *decorator* que não interfere na função aplicada.

4.8.5 Compatibilidade completa com a linguagem

É provável que a maioria das tentativas de otimização falhem, visto que pouquíssimos recursos da linguagem foram implementados. Porém, graças às características anteriores, o interpretador é capaz de executar o código não reconhecido pelo otimizador. Assim, a compatibilidade não é quebrada – o mesmo não acontece com ***Cython*** e ***Shed Skin***, por exemplo, como visto no capítulo 3.

5 RESULTADOS

A análise comparou o desempenho desta implementação em relação a códigos gerados para:

- o interpretador ***CPython*** "puro" (versão 3.3.2)
- o interpretador ***PyPy*** "puro" (versão 2.2.1, com JIT)
- a biblioteca ***Cython*** (versão 0.19.2)
- a biblioteca ***Numba*** (versão 0.11.0)

A comparação foi realizada com a utilização de três algoritmos:

1. Algoritmo de Dijkstra [69, Capítulo 24: *Single-Source Shortest Paths*, pp. 595-601], utilizando um *heap* binário [69, Capítulo 6: *Heapsort*, pp. 127-135] como fila de prioridade.
2. Produto de matrizes [69, Capítulo 25: *All-Pairs Shortest Paths*, pp. 625], computado pelo algoritmo *naïve*, com complexidade $O(n^3)$. A fim de valorizar as localidades espacial e temporal, a transposição da segunda matriz é realizada antes da execução do algoritmo.
3. Algoritmo *Quicksort* [69, Capítulo 7: *Quicksort*, pp. 145-164].
4. *Pidigits* [5], *benchmark* que computa o valor de π com n dígitos, sendo n o argumento de entrada.

Para os três primeiros testes, foram utilizadas entradas razoavelmente grandes, com o objetivo de minimizar o impacto das conversões de tipos (de CPython para C++ na entrada e de C++ para CPython na saída).

O *benchmark Pidigits* não realiza conversões de tipos, visto que o argumento de entrada é apenas um inteiro.

Cada algoritmo de teste foi executado nove vezes em cada ambiente (interpretador ou biblioteca). A tabela 5.1 contém o tempo médio de execução em segundos (t_m) e o coeficiente de variação dos tempos (C_v), para cada algoritmo de teste.

	Produto de matrizes		***Quicksort***		**Dijkstra**		***Pidigits***	
	t_m	C_v	t_m	C_v	t_m	C_v	t_m	C_v
CPython	$24,0$	$0,018$	$40,3$	$0,010$	$7,15$	$0,001$	$5,39$	$0,008$
Meu trabalho	$0,40$	$0,011$	$0,81$	$0,006$	$1,74$	$0,004$	$5,15$	$0,017$
PyPy	$3,43$	$0,003$	$2,10$	$0,022$	$3,39$	$0,059$	$7,96$	$0,004$
Cython	$13,0$	$0,050$	$1,67$	$0,009$	$1,97$	$0,002$	$1,02$	$0,007$
Numba	–	–	–	–	–	–	–	–

Tabela 5.1: Média e coeficiente de variação dos tempos absolutos de execução, em segundos. Quanto menor o valor, melhor o desempenho.

Os valores contidos na tabela 5.2 demonstram a média do ganho de desempenho em relação ao CPython, para cada algoritmo de teste.

A biblioteca ***Numba*** não foi capaz de compilar o código de nenhum dos testes propostos. Tal problema pode ser fundamentado por ser um projeto recente.

Os testes mostram que a implementação proposta alcança uma melhora expressiva no desempenho em relação ao código puramente interpretado (cerca de 50x para o produto de matrizes e para o *Quicksort*).

O ganho reduzido para o algoritmo de Dijkstra pode ser justificado pelo uso de um *heap* implementado em C, disponibilizado na biblioteca padrão do CPython.

Não houve ganho no *benchmark Pidigits*, devido ao alto custo das operações aritméticas sobre inteiros de precisão arbitrária. Neste *benchmark*, a biblioteca ***Cython*** obteve um resultado superior, justificado pelo uso da biblioteca GMP [70].

	Produto de matrizes	***Quicksort***	**Dijkstra**	***Pidigits***
CPython	$1,0$	$1,0$	$1,0$	$1,0$
Meu trabalho	$59,7$	$50,0$	$4,1$	$1,0$
PyPy	$7,0$	$19,3$	$2,1$	$0,7$
Cython	$1,9$	$24,2$	$3,6$	$5,3$
Numba	–	–	–	

Tabela 5.2: Comparação de desempenho entre as alternativas propostas. Valores normalizados em relação ao CPython. Quanto maior o valor, melhor o desempenho.

6 CONCLUSÃO E TRABALHOS FUTUROS

O grande número de alternativas disponíveis para contornar o problema de desempenho de Python, algumas inclusive comerciais, demonstram a relevância do assunto. Os excelentes resultados obtidos justificam e criam espaço para trabalhos futuros.

Porém, mesmo sendo uma linguagem bastante simples, é difícil alcançar uma cobertura ampla dos seus recursos.

6.1 Trabalhos futuros

Inicialmente, era do meu interesse que este projeto utilizasse LLVM [71] como representação intermediária, devido às possibilidades providas (como JIT), à crescente popularidade da plataforma e pelo aprendizado necessário.

LLVM exige que o código esteja na forma *Static Single Assignment* [72] (SSA é também um elemento importante de bibliotecas como ***Blaze*** [40], ***Numba*** [73] e ***Parakeet*** [41], citadas neste trabalho, e também por compiladores como o *Intel C++ Compiler*[1]). Por estas razões, SSA é um caminho a ser fortemente considerado.

Outro ponto importante para o futuro é a inferência de tipos. Embora cada nodo de *expr* (como *BinOp* e *Compare*) possua um tipo na representação interna dos estágios, este trabalho não teve a finalidade de realizar inferência automática de tipos (mesmo quando esta é trivial). A inserção deste recurso para casos simples demanda poucas alterações no projeto.

Seria também importante a implementação de um recurso para liberar temporariamente o *Global Interpreter Lock* (recurso permitido por bibliotecas como ***Cython***), o que permitiria atingir paralelismo real no CPython..

Além destes, os seguintes tópicos surgiram durante a execução deste trabalho:

- Implementar alguns recursos da linguagem listados nas limitações.
- Analisar a arquitetura de soluções similares.
- Realizar a verificação de erros semânticos.
- Permitir a criação de classes otimizadas.

[1] *"The Intel compiler is a mature optimizing compiler that translates C++, C, and C99, supporting all the standard optimizations. Since it has used static single assignment (SSA) form for over 10 years, the compiler efficiently performs conditional constant propagation, partial redundancy elimination (PRE), and common subexpression elimination (CSE)."* [74]

BIBLIOGRAFIA

[1] *Mozilla Labs : PDF.js*. URL: https://mozillalabs.com/en-US/pdfjs/.

[2] *Shumway*. URL: http://mozilla.github.io/shumway/.

[3] *mbebenita/Broadway · GitHub*. URL: https://github.com/mbebenita/broadway.

[4] *The Julia Language*. URL: http://julialang.org/.

[5] *Python 3 vs C++ g++ | Computer Language Benchmarks Game*. URL: http://benchmarksgame.alioth.debian.org/u32/benchmark.php?test=all&lang=python3&lang2=gpp&data=u32.

[6] *Python Programming Language – Official Website*. [Accessado em 05/12/2013]. URL: http://python.org/.

[7] *The Jython Project*. URL: http://www.jython.org/.

[8] *IronPython - Home*. URL: http://ironpython.codeplex.com/.

[9] *Dynamic Language Runtime - Home*. URL: http://dlr.codeplex.com/.

[10] *PyPy - Welcome to PyPy*. URL: http://pypy.org/.

[11] *Getting Started with RPython — PyPy 2.2.1 documentation*. URL: http://doc.pypy.org/en/latest/getting-started-dev.html#getting-started-with-rpython.

[12] *What's New In Python 3.0 — Python v3.3.3 documentation*. URL: http://docs.python.org/3.3/whatsnew/3.0.html.

[13] *DistroWatch.com: Put the fun back into computing. Use Linux, BSD.* URL: http://distrowatch.com/search.php?pkg=Python&pkgver=2.

[14] *Python/C API Reference Manual — Python v3.3.3 documentation*. URL: http://docs.python.org/3.3/c-api/.

[15] *1. Extending Python with C or C++ — Python v3.3.3 documentation*. URL: http://docs.python.org/3.3/extending/extending.html.

[16] *Introduction — Python v3.3.3 documentation*. URL: http://docs.python.org/3.3/c-api/intro.html.

[17] Daniel C. Wang et al. "The Zephyr Abstract Syntax Description Language". Em: *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*. DSL'97. Berkeley, CA, USA: USENIX Association, 1997, pp. 213–228. URL: http://dl.acm.org/citation.cfm?id=1267950.1267967.

[18] *Glossary — Python v3.3.3 documentation*. URL: http://docs.python.org/3.3/glossary.html#term-global-interpreter-lock.

[19] *GlobalInterpreterLock - Python Wiki*. URL: https://wiki.python.org/moin/GlobalInterpreterLock.

[20] *Glossary — Python v3.3.3 documentation*. URL: http://docs.python.org/3.3/glossary.html#term-decorator.

[21] *PEP 3107 – Function Annotations*. URL: http://www.python.org/dev/peps/pep-3107/.

[22] *haypo / astoptimizer — Bitbucket*. URL: https://bitbucket.org/haypo/astoptimizer/.

[23] *16.17. ctypes — A foreign function library for Python — Python v3.3.3 documentation*. URL: http://docs.python.org/3.3/library/ctypes.html.

[24] *CFFI documentation — CFFI 0.7.2 documentation*. URL: http://cffi.readthedocs.org.

[25] *Simplified Wrapper and Interface Generator*. URL: http://www.swig.org/.

[26] *www*. URL: http://www.f2py.com/.

[27] *Boost.Python - 1.54.0*. URL: http://www.boost.org/libs/python.

[28] *Babel Homepage*. URL: http://computation.llnl.gov/casc/components/#page=home.

[29] *Riverbank | Software | SIP | What is SIP?* URL: http://www.riverbankcomputing.com/software/sip/intro.

[30] *pyfortran.sourceforge.net/*. URL: http://pyfortran.sourceforge.net/.

[31] *fenics-project / Instant — Bitbucke*. URL: https://bitbucket.org/fenics-project/instant.

[32] *Automated Solution of Differential Equations by the Finite Element Method — FEniCS Project*. URL: http://fenicsproject.org/.

[33] *PyInline: Mix Other Languages directly Inline with your Python*. URL: http://pyinline.sourceforge.net/.

[34] *Weave (scipy.weave) — SciPy v0.13.0 Reference Guide*. URL: http://docs.scipy.org/doc/scipy/reference/tutorial/weave.html.

[35] *SciPy.org — SciPy.org*. URL: http://www.scipy.org/.

[36] *NumPy — Numpy*. URL: http://www.numpy.org/.

[37] *Pythran*. URL: http://pythonhosted.org/pythran/.

[38] *shedskin - An experimental (restricted-Python)-to-C++ compiler - Google Project Hosting*. URL: http://code.google.com/p/shedskin/.

[39] *unpython - unPython : Python to C compiler - Google Project Hosting*. URL: http://code.google.com/p/unpython/.

[40] *Blaze - Continuum Analytics*. URL: http://blaze.pydata.org/.

[41] *Parakeet: a runtime compiler for numerical Python*. URL: http://www.parakeetpython.com/.

[42] *Copperhead*. URL: http://copperhead.github.io/.

[43] *OpenMP.org*. URL: http://openmp.org/.

[44] *Home | Threading Building Blocks*. URL: https://www.threadingbuildingblocks.org/.

[45] *Welcome — Theano 0.6 documentation*. URL: http://deeplearning.net/software/theano/.

[46] *markflorisson88/minivect · GitHub*. URL: https://github.com/markflorisson88/minivect.

[47] *numexpr - Fast numerical array expression evaluator for Python and NumPy. - Google Project Hosting*. URL: http://code.google.com/p/numexpr/.

[48] *Vector Math Library (VML) Performance and Accuracy Data*. URL: http://software.intel.com/sites/products/documentation/hpc/mkl/vml/vmldata.htm.

[49] *PythonFrontEnd - GCC Wiki*. URL: http://gcc.gnu.org/wiki/PythonFrontEnd.

[50] *Psyco - Home Page*. URL: http://psyco.sourceforge.net/.

[51] *unladen-swallow - A faster implementation of Python - Google Project Hosting*. URL: http://code.google.com/p/unladen-swallow/.

[52] *2c-python - 2C.py - Python-to-C compiler for CPython 2.6 - Google Project Hosting*. URL: http://code.google.com/p/2c-python/.

[53] *Nuitka Home | Nuitka Home*. URL: http://nuitka.net/.

[54] *mypy - A New Python Variant with Dynamic and Static Typing*. URL: http://www.mypy-lang.org/.

[55] *Faster code via static typing — Cython 0.19.2 documentation*. URL: http://docs.cython.org/src/quickstart/cythonize.html.

[56] *NumbaPro — Continuum documentation*. URL: http://docs.continuum.io/numbapro/.

[57] *Information technology – Programming languages – C++*. Standard. 2011.

[58] *PEP 3131 – Supporting Non-ASCII Identifiers*. URL: http://www.python.org/dev/peps/pep-3131/.

[59] *Unicode Consortium*. URL: http://www.unicode.org/.

[60] *PEP 20 – The Zen of Python*. URL: http://www.python.org/dev/peps/pep-0020/.

[61] *PEP 255 – Simple Generators*. URL: http://www.python.org/dev/peps/pep-0255/.

[62] *PEP 342 – Coroutines via Enhanced Generators*. URL: http://www.python.org/dev/peps/pep-0342/.

[63] *PEP 289 – Generator Expressions*. URL: http://www.python.org/dev/peps/pep-0289/.

[64] *PEP 343 – The "with"Statement*. URL: http://www.python.org/dev/peps/pep-0343/.

[65] *PEP 202 – List Comprehensions*. URL: http://www.python.org/dev/peps/pep-0202/.

[66] *PEP 274 – Dict Comprehensions*. URL: http://www.python.org/dev/peps/pep-0274/.

[67] *PEP 3115 – Metaclasses in Python 3000*. URL: http://www.python.org/dev/peps/pep-3115/.

[68] *5. The import system — Python v3.3.3 documentation*. URL: http://docs.python.org/3.3/reference/import.html.

[69] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd. McGraw-Hill Higher Education, 2001. ISBN: 0070131511.

[70] *The GNU MP Bignum Library*. URL: https://gmplib.org/.

[71] *The LLVM Compiler Infrastructure Project*. URL: http://llvm.org/.

[72] Ron Cytron et al. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". Em: *ACM Trans. Program. Lang. Syst.* 13.4 (out. de 1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320. URL: http://doi.acm.org/10.1145/115372.115320.

[73] *Numba — numba*. URL: http://numba.pydata.org/.

[74] *High-Performance Computing with Intel® C++ and Intel® VTune™ | Intel® Developer Zone*. URL: http://software.intel.com/en-us/articles/high-performance-computing-with-intel-c-and-intel-vtunet.

Apêndice A

A.1 Funções da Python/C API que tomam a referência dos objetos passados

Esta lista, compilada manualmente de dados obtidos da documentação oficial, contém todas as funções da Python/C API da versão 3.3 do CPython que tomam a referência dos objetos passados como argumentos:

- **void PyErr_SetExcInfo(PyObject *type, PyObject *value, PyObject *traceback)**: toma a referência dos objetos passados em todos os argumentos.
- **void PyException_SetContext(PyObject *ex, PyObject *ctx)**: toma a referência de **ctx**.
- **void PyException_SetCause(PyObject *ex, PyObject *ctx)**: toma a referência de **ctx**.
- **int PyTuple_SetItem(PyObject *p, Py_ssize_t pos, PyObject *o)**: toma a referência de **o**.
- **void PyTuple_SET_ITEM(PyObject *p, Py_ssize_t pos, PyObject *o)**: toma a referência de **o**.
- **int PyList_SetItem(PyObject *list, Py_ssize_t index, PyObject *item)**: toma a referência de **item** e descarta a referência do objeto que estava previamente em `list[index]`.
- **void PyList_SET_ITEM(PyObject *list, Py_ssize_t i, PyObject *o)**: toma a referência de **item** mas, ao contrário de **PyList_SetItem**, não descarta a referência do objeto que estava previamente em `list[index]`.
- **int PyModule_AddObject(PyObject *module, const char *name, PyObject *value)**: toma a referência de **value**.

A.2 ASDL

A seguinte ASDL é idêntica à utilizada pelo Python 3.3, variando apenas no espaçamento.

```
-- ASDL's five builtin types are identifier,
                                 int,
                                 string,
                                 bytes,
```

object

```
module Python
{
    mod = Module(stmt* body)
        | Interactive(stmt* body)
        | Expression(expr body)

        -- not really an actual node but useful in
        -- Jython's typesystem.
        | Suite(stmt* body)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list,
                       expr? returns)
          | ClassDef(identifier name,
             expr* bases,
             keyword* keywords,
             expr? starargs,
             expr? kwargs,
             stmt* body,
             expr* decorator_list)
          | Return(expr? value)

          | Delete(expr* targets)
          | Assign(expr* targets, expr value)
          | AugAssign(expr target, operator op, expr value)

          -- use 'orelse' because else is a keyword in
          -- target languages
          | For(expr target, expr iter, stmt* body,
                stmt* orelse)
          | While(expr test, stmt* body, stmt* orelse)
          | If(expr test, stmt* body, stmt* orelse)
          | With(withitem* items, stmt* body)

          | Raise(expr? exc, expr? cause)
          | Try(stmt* body, excepthandler* handlers,
                stmt* orelse, stmt* finalbody)
          | Assert(expr test, expr? msg)

          | Import(alias* names)
          | ImportFrom(identifier? module, alias* names,
                       int? level)

          | Global(identifier* names)
          | Nonlocal(identifier* names)
          | Expr(expr value)
```

```
       | Pass | Break | Continue

       -- XXX Jython will be different
       -- col_offset is the byte offset in the
       -- utf8 string the parser uses
       attributes (int lineno, int col_offset)

       -- BoolOp() can use left & right?
expr = BoolOp(boolop op, expr* values)
     | BinOp(expr left, operator op, expr right)
     | UnaryOp(unaryop op, expr operand)
     | Lambda(arguments args, expr body)
     | IfExp(expr test, expr body, expr orelse)
     | Dict(expr* keys, expr* values)
     | Set(expr* elts)
     | ListComp(expr elt, comprehension* generators)
     | SetComp(expr elt, comprehension* generators)
     | DictComp(expr key, expr value,
                comprehension* generators)
     | GeneratorExp(expr elt,
                    comprehension* generators)
     -- the grammar constrains where
     -- yield expressions can occur
     | Yield(expr? value)
     | YieldFrom(expr value)
     -- need sequences for compare to
     -- distinguish between
     -- x < 4 < 3 and (x < 4) < 3
     | Compare(expr left, cmpop* ops,
               expr* comparators)
     | Call(expr func, expr* args, keyword* keywords,
         expr? starargs, expr? kwargs)
     | Num(object n) -- a number as a PyObject.
     | Str(string s) -- need to specify raw,
                     -- unicode, etc?
     | Bytes(bytes s)
     | Ellipsis
     -- other literals? bools?

     -- the following expression can appear in
     -- assignment context
     | Attribute(expr value, identifier attr,
                 expr_context ctx)
     | Subscript(expr value, slice slice,
                 expr_context ctx)
     | Starred(expr value, expr_context ctx)
     | Name(identifier id, expr_context ctx)
     | List(expr* elts, expr_context ctx)
```

```
      | Tuple(expr* elts, expr_context ctx)

       -- col_offset is the byte offset in the
       -- utf8 string the parser uses
       attributes (int lineno, int col_offset)

expr_context = Load | Store | Del | AugLoad
             | AugStore | Param

slice = Slice(expr? lower, expr? upper, expr? step)
      | ExtSlice(slice* dims)
      | Index(expr value)

boolop = And | Or

operator = Add | Sub | Mult | Div | Mod | Pow
         | LShift | RShift | BitOr | BitXor
         | BitAnd | FloorDiv

unaryop = Invert | Not | UAdd | USub

cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is
      | IsNot | In | NotIn

comprehension = (expr target, expr iter, expr* ifs)

excepthandler = ExceptHandler(expr? type,
                              identifier? name,
                              stmt* body)
                attributes (int lineno, int col_offset)

arguments = (arg* args,
             identifier? vararg,
             expr? varargannotation,
             arg* kwonlyargs,
             identifier? kwarg,
             expr? kwargannotation,
             expr* defaults,
             expr* kw_defaults)
arg = (identifier arg, expr? annotation)

-- keyword arguments supplied to call
keyword = (identifier arg, expr value)

-- import name with optional 'as' alias.
alias = (identifier name, identifier? asname)

withitem = (expr context_expr, expr? optional_vars)
```

```
}
```

A.3 Especificação da linguagem

Esta seção tem o objetivo de especificar o subconjunto da linguagem Python suportado pelo otimizador

A.3.1 Tipos nativos

Os seguintes tipos podem ser utilizados na anotação das variáveis locais:

- bool
- float32
- float64
- int32
- int64
- size_t
- string
- *Containers*
  - pair
  - priority_queue
  - vector

A.3.2 Estruturas de controle de fluxo

Atualmente, as seguintes estruturas de controle de fluxo são suportadas:

- if
- if/else
- if/elif/else
- for
- for/else [1]
- while
- while/else [1]
- return

As estruturas que seguem não são atualmente suportadas:

- *Generators*
- raise
- try/except
- try/finally
- try/except/finally

A.3.3 Metaprogramação

O uso das funções `compile`, `eval` e `exec` não é suportado.

[1] Em Python, **`for`** e **`while`** possuem uma variação com **`else`**, cujo código é executado quando o laço termina sem o uso de **`break`**.

A.3.4 *Comprehensions*

O uso de *dict [66]/list [65]/set comprehensions* não é suportado atualmente.

A.3.5 *Builtins*

Atualmente, a únicas funções nativas suportadas são `divmod`, `len` e `range`. O uso de outras funções é intermediado pelo interpretador.

A.3.6 Operadores

Todos os operadores da linguagem são suportados.

A.3.7 Funções matemáticas

- abs
- round
- Operações de ponto flutuante (definidas no módulo ***math***)
  - acos
  - acosh
  - asin
  - asinh
  - atan
  - atan2
  - atanh
  - ceil
  - copysign
  - cos
  - cosh
  - erf
  - erfc
  - exp
  - expm1
  - fabs
  - floor
  - fmod
  - frexp
  - gamma
  - hypot
  - isfinite
  - isinf
  - isnan
  - ldexp
  - lgamma
  - log
  - log10
  - log1p
  - modf
  - pow

- sin
- sinh
- sqrt
- tan
- tanh
- trunc