Konohaにおける静的スクリプティングの文法設計

倉光君郎

Konoha は，スクリプティング言語に静的な型付けを導入した独自設計の言語である．従来の動的なスクリプティングと同様な「プログラミングしやすさ」を目指している．一方，静的な型付けによってソースコードには型情報が付加され，これらを有効に活用したスタイルのスクリプティングも可能になる．本稿では，Konoha の静的スクリプティングの側面に焦点をあて，型推論の導入からリソースパス，型変換推論などの言語設計を報告する．

1 はじめに

スクリプティング言語は，ある応用領域における「プログラミングしやすさ」を最大化するように設計されて，オペレーティングシステムの運用やアプリケーションシステムの再構成などの用途で広く利用されてきた．近年，オブジェクト指向スクリプティング言語の登場により，汎用的なプログラミング能力は大きく向上し，Web 応用からゲーム開発までさまざまな分野への採用が広がっている．

我々は，2006 年以来，新しく静的に型付けされたスクリプティング言語，Konoha の設計と実装に取り組んでいる．Konoha の目標は，スクリプティングの世界に静的言語の技法，つまりコンパイル時の型検査による実行前のソースコード検証，静的に決定された型情報による高速な実行環境を導入し，スクリプティング言語の新たな可能性を開拓することにある．同時に，動的言語のプログラミング経験の再現など「プログラミングしやすさ」の維持も図っている．

これらの目標は,「スクリプティングとは動的型付けである」という前提で，静的な型付けと動的言語の融合の試みといえる．他方で，静的な型付けは，言語処理系が機械処理できる付加的な情報を加えることになる．我々は，これらの付加された型情報によって，動的言語とは異なる「プログラミングしやすさ」を実現できると期待している．我々は，型を利用したスクリプティング能力の向上を目指し，本論文では，Konoha 言語における「静的スクリプティング」のための文法設計を報告する．

まず，Konoha 言語を簡単に紹介する．Konoha は，C 言語上でスクラッチから開発されたオブジェクト指向スクリプティング言語である．スクリプトは静的に型付けされ，専用のバイトコードにコンパイルされ，我々が開発した Konoha 仮想マシン (VM) の上で動作される．

Konoha の大きな特徴は，スクリプト実行前に，コンパイラによって型エラーが検出され，それらがエラー報告 (図 1) として出力される．同時に，エラーが発見されたコードは，実行例外を throw するコードに書き換えられ，実行可能なバイトコードが生成される．これにより，従来からのスクリプティング言語と同様に，全てのエラーを取り除かなくても試しに実行することができる．(コンパイラによるエラー検証のみおこなうモードもある.)

一方，Konoha は，スクリプティング言語の特徴的機構である動的な評価 (eval) によるプログラム更新と実行が可能である．加えて，Python スタイルの対

JST / CREST DEOS プロジェクト
横浜国立大学, Yokohama National University

```
$ cat tarai_bugs.k
#!/usr/local/konoha
/* tarai.k */
int tarai(int x, int y, int z) {
  if (x > y)
    return tarai(tarai(x - 1, y, z), tarai(y - 1, z, x), tarai(z - 1, x, y));
  else
    return y;
}
print tarai("this is wrong", a);
print tara1(12, 6, 0);

$ konoha tarai_bugs.k
 - [tarai_bugs.k:11]:(type error) parameter 1 of tarai must be int , not String
 - [tarai_bugs.k:11]:(error) undefined variable: a
 - [tarai_bugs.k:12]:(error) undefined method: Script.tara1
```

図 1　Konoha における静的検査の実行

話シェルを備え，プログラムを対話的に動作させることができる．図 2 は，対話シェルによるプログラムの実行例である．

本論文では，Konoha の振る舞いを表記するため，対話モードによる実行結果を用いる．>>>は，式やステートメントを入力するプロンプトであり，その評価結果は次の行に表示される．

```
>>> tarai(12, 6, 0)     // 式の入力
12                  // 式の評価結果
```

Konoha は，過去から現在にわたり，さまざまな設計変更がおこなわれてきた．本論文は，ソフトウェア論文として，デザイン選択に焦点をおいた議論をおこない，対比のために実際に現在は動作しないサンプル記述も掲載する．Konoha 言語は，次のサイトからオープンソースとして公開され，もっとも初期バージョンから最新版まで，svn レポジトリからソースコードを入手することができる．

```
$ svn checkout
  http://svn.sourceforge.jp/svnroot/konoha
```

本論文の構成は，以下のとおりである．第 2 節では Konoha の基本的な文法設計を紹介する．第 3 節では，型宣言の省略から Konoha で導入された型推論機構を述べる．第 4 節では，型変換推論，リソースパス，文法とクラス設計を融合させた議論をおこなう．第 5 節では，本論文を総括する．

2　Konoha の言語設計

プログラミング言語は，技術である前に「ことば」である．プログラミングの生態系を考えたとき，個別の言語自体より,「ことば」を「話」しているユーザ層が大きな部分を占める．この「ことば」という性質を考えると，全く新しい言語文法を導入することは難しく，既存の「ことば」の話者が混乱させない配慮が必要となる．本稿では，互換性, 拡張性, 独自性という観点から，Konoha の言語設計を述べる．

2.1　標準語

プログラミング言語は,「ことば」である．現在，C 言語から始まり，C++, Java, C#と続く，言語文法の系統は,「ことば」としての主流といえる．我々は，

```
$ konoha    ## 対話モードで起動
Konoha 0.7(aomori) source (rev:1762, Aug  3 2010 07:08:43)
[GCC 4.2.1 (Apple Inc. build 5659)] on macosx_64 (64, UTF-8)
Options: rcgc thread used_memory:351 kb

>>> int tarai(int x, int y, int z) {
  if (x > y) return tarai(tarai(x - 1, y, z), tarai(y - 1, z, x), tarai(z - 1, x, y));
  else return y;
}
>>> tarai(12, 6, 0)
12
>>> tarai(14, 7, 0)
14
>>>
```

図 2　Konoha 対話シェル: tarai 関数の定義と対話的な実行の様子

表 1　Konoha 0.7 ステートメント: * は，Konoha で独自のセマンティクスをもつ

種類	ステートメント
名前空間	namespace* using* include*
分岐	if/else switch/case
ループ	while do/while for foreach*
例外	try/catch/finally
ジャンプ	break continue return throw
クラス	class/extends
クロージャ	function*
デバッグ	print* assert*

特に Java 言語を「標準語」と考え，文法と語彙（クラス，メソッド）を可能な限り継承する方針で Konoha の言語設計をおこなっている．

表 1 は、Konoha のステートメント一覧である．* がついたステートメントは，Konoha 独自のセマンティクスが定義されている．制御構造や例外処理，クラス定義など，基本的にプログラミング言語として確立され，Konoha において新規性を提供しない場合は，同じステートメントを採用し，同じ振る舞いを実現している．そのため，Konoha は，静的言語として変数宣言をおこなう点も加味すると，「Java ライク」と冠される既存の言語と比べて，より高いレベルのソースコード互換性をもっている．次は，Java 言語と完全にソースコード互換な Counter クラスの例である．

```
class Counter {
  int _counter;
  Counter(int counter) {
    _counter = counter;
  }
  void count() {
    _counter++;
  }
}

Counter c = new Counter(0);
c.count();
```

さらに，言語の類似性を考えるとき，単に文法構造が近いだけでなく，語彙も近いことが重要になる．プログラミング言語において，語彙はクラスやメソッド名に相当する．表 2 は，Konoha でサポートしている基本クラスの一覧である．Java 言語は，大規模なクラスライブラリを誇っており，我々は全てのクラスを再定義することはできないが，Java 言語に由来するクラスの場合は，そのフィールドやメソッド名も含めて，可能な限り忠実にクローンしている．

表 2　Konoha 0.7 基本クラス

種類	クラス名
Top	Object
Data Value	Boolean *Number* Int Float String Bytes
Collection	Array<T> Iterator<T> Tuple Map <K,V>
Language	Class Method Func Translater System Script Context NameSpace Exception
I/O	InputStream OutputStream Connection ResultSet
Misc	Regex Converter
Bottom	Any

次は，Java における print 文であるが，Konoha でも同じそのまま解釈して動作することができる．

```
System.out.println("hello,world\n");
```

2.2　拡張

豊富な演算子の提供は，スクリプティング言語の特徴である．Konoha では，演算子は文法の一部とみなし，文法と同様に Java 言語が提供する演算子を忠実に再現する方針をとっている．例えば，Konoha は純粋なオブジェクト指向であり，整数は不変なオブジェクトとして扱われるが，前置 `++i` と後置 `i++`区別を区別して，インクリメンタル演算子をエミュレーションしている．

一方,「スクリプトらしさ」を実現するためには，Python や JavaScript などで広く使われている演算子を提供することが重要である．Konoha では，Java 言語の演算子セマンティクスを維持しながら，新しい演算子を定義する方針を採用している．

- **Java から継承した演算子．**条件演算子 (`&&`, `||`, `!`), 三項演算子 (`? :`)，比較演算子 (`==`, `!=`, `<`, `<=`, `>`, `>=`)，型判定演算子 (`instanceof`)，四則演算子 (`+`, `-`, `*`, `/`, `\%`)，インクリメンタル演算子 (`++`, `--`) ビット演算子 (`<<`, `>>`, `|`, `&`/, `^`, `~`)
- **読みやすさのための別名．**条件演算子 (`and`, `or`, `not`)，四則演算子 (`mod`),
- **他言語からの借用．**包含演算子 (`in?`), Null 演算子 (`??`), スライス演算子 (`[m..n]`, `[m to n]`, `[m until n]`)，パターン演算子 (`=~`) など

Konoha では，上述以外にも，いくつかの独自の演算子が実験的に拡張されている．

Java 意味論との唯一の非互換部分は，比較演算子 (`==`, `!=`, `<`, `<=`, `>`, `>=`) である．Java では，オブジェクト参照の比較にこれらの演算子を用いていたが，Konoha では（もし同じ型の比較であれば）オブジェクトの値を比較するように変えている．これにより，文字列の比較を直接，比較演算子でおこなえるようになっている．

```
>>> "naruto" == "NARUTO"
false
>>> "naruto" > "NARUTO"
true
```

Konoha では，演算子の多重定義を抑制している．後述するとおり，型推論システムの導入を検討しているため，無秩序に演算子の多重定義を認めると，型推論時の決定性が下がるためである．例えば，`|s|`は，シーケンス構造 `s` からその大きさを取り出す演算子であるが，これは整数の絶対値を得る演算子として多重定義をおこなうと，候補が増えるため型推論の処理時間が大きくなるためである．

2.3　独自の拡張

Konoha は，独自のリテラル，ステートメント，演算子など，さまざまな言語機能を追加している．本節では，静的なスクリプティングに強く関連するリソースパスに関して説明する．

プログラミング言語は，プログラムの内部状態に対し，名前（変数名や関数名）など，名前をつけて扱ってきた．しかし，コンピュータシステムによる情報空間がネットワーク化されている現在，プログラミング言語で扱うべき情報の多くは，コンピュータシステム外部にある．外部に存在するリソースとして代表例は，ファイル，データベース，Web リソースがある．

Konoha では，言語文法レベルにおいて，これらの外部リソースを直接扱う識別子，リソースパスを導

入している．リソースパスは，URL などでよく知られた URN(universal resource name) の記法を用い，リソースの種類とそのパスを表している．

```
http://www.w3.org/
file:/etc/passwd
lib:iconv
ip:133.33.31.12
```

リソースパスは，Boolean 型に型付けられている．コンパイル時に静的にその存在が確認され，定数として畳み込まれる．

```
>>> file:/etc/passwd
true
>>> dir:/etc/passwd
false
```

一方，実行時に評価したい場合は，文字列に対してパス修飾子を用いる．パス修飾子は，リソースの種類を表す部分にコロンを追加した記法である．プログラマは，文字列をどのような種類のパスとして扱うつもりなのか，静的に与えなければならない．

```
>>> pwd = "/etc/passwd"
>>> file:: pwd
true
```

リソースパスは，単純に外部リソースの存在を判定するだけではなく，リソース本体に対するアクセスにも利用できる．さまざまな種類の異なるリソースへのアクセスを実現するため，我々は静的スクリプティングを提案する．この手法に関しては，第 4 章で詳しく述べる．

3 型宣言の省略と型推論

プログラマからみたとき，静的言語と動的言語の大きな違いは，明示的な型宣言の有無にある．近年，OCaml や Haskel など，型推論を備えた静的言語が普及し，動的言語とよく似たプログラミング経験が実現されることが知られている．しかし，型推論と動的な型付けの意味論は全く異なる．本節では，静的スクリプティングを論じる出発地点として，型推論の導入を通した静的スクリプティングを論じる．

3.1 背景

スクリプティング言語は，ラピッドプロトタイピング用途に用いられることが多い．この際，自明な変数宣言を強要することは予想外に煩わしい．Konoha では，スクリプトらしさを実現するため，「変数代入による初期化」によって暗示的な変数宣言[†1]とみなし，自明な変数宣言を省略可能にした．

```
a = [0, 1, 2]     // 変数 a の初期化
a.add(3)
```

Konoha は，静的言語であるため，暗黙的に宣言された変数であってもコンパイル時に型を決定する必要がある．初期の言語設計 [6] では，動的言語の振る舞いを忠実に再現しようと考えていたため，Object 型を型システムのトップとしたときのボトムという意味で Any 型を導入した．これは，C#4.0 の dynamic 型とほぼ等しく，コンパイル時には型検査を行わず，実行時に型検査を加える特別な型である．

次は，初期の Konoha における変数 a の暗示的な宣言の実行例である．ここで，変数 a は，Any 型として，任意の型が代入可能になる．

```
 >>> a = [0, 1, 2]     // 初期化
 >>> a.class    // オブジェクトの型
int[]
>>> typeof(a) // 変数の型
Any
>>> a[0]
0
>>> a = "naruto"    // 変数を再現
>>> a.class
String
>>> a[0]
"n"
```

Duck typing は，動的言語において重要なしかも便利なプログラム抽象化手段として知られている．これは動的型付けにおける実行時型検査の特性を活かし，厳密に抽象クラス等で定義することなくよく似たオブジェクトを同じコードで扱うことを可能にする．例えば，上述の変数 a では，抽象的なシーケンス型

†1 変数宣言もしくは初期化されていない変数は未定義の変数としてエラーとなる．

として型付けすることなく，共通する演算子であるインデックサ a[0] を扱うコードになっている．Any 型は，このような duck typing を可能にする．

3.2 変数の型推論

Konoha[6] は，Any 型を導入することで変数宣言を省略可能にし，動的言語の振る舞いを忠実に再現した．しかし，初期の利用経験から，変数宣言が省略できる場合，プログラマはほとんど変数宣言をおこなわないことがわかってきた．この場合，ほとんどの変数が Any 型として扱われ，Konoha の静的言語としての型検査機能があまり有効とならない．同時に，実行時の型検査による処理のオーバヘッドも無視できなくなった．

Konoha 0.2 バージョンから，初期化における右辺値から変数の型を推論する方式を採用した．

```
>>> a = [0, 1, 2]     // 初期化
>>> a.class    // オブジェクトの型
int[]
>>> typeof(a) // 変数の型
int[]
```

変数 a の型を気にしなければ，型推論も Any 型も見かけ上ほとんど変わらない．しかし，前述の例では，変数 a は，Any に代わり int[] と厳密に決定されるため，整数配列以外は型エラーとして代入することができなくなる．

```
>>> a = [0, 1, 2]     // 初期化
...
>>> a = "NARUTO"
[(eval):1]:(type error) must be Int[],
   not String
```

問題は，静的に型を決定して不都合は生じないかという点である．Konoha は静的言語としてスクリプトの機械検査を目指しているため，むしろ望ましいと考えている．さらに，JavaScript のソースコード解析?[] によると，動的言語であっても実行時に型が変わるソースコードは稀であると報告されており，我々は型推論による静的に型を決定しても大きく書きやすさを損なわないと考えている．唯一の例外として，duck typing がある．型推論では，曖昧な類似性から抽象的な型を推論することが難しい．そのため，明示的に Any 型を宣言することで，duck typing スタイルのプログラミングも可能にしている．

```
>>> Any a = [0, 1, 2];
>>> a[0]
0
>>> a = "naruto"
>>> a[0]
"n"
```

3.2.1 パラメータの型推論

Konoha は，静的言語であるため，通常，メソッドや関数を定義するとき，パラメータや戻り値の型を明示的に与える必要がある．

```
int fibo(int n) {
  if(n < 3) return n;
  return fibo(n-1) + fibo(n-2);
}
```

動的言語は，一方，型付けをおこなわないため，パラメータや戻り値の型も省略して定義する．つまり，上記の fibo 関数は，次のとおりになる．

```
function fibo(n) {
  if(n < 3) return n;
  return fibo(n-1) + fibo(n-2);
}
```

このとき，型推論を用いれば，パラメータや戻り値の型[†2]も静的に決定することができる．Konoha の過去のいくつかのバージョンでは，試験的にパラメータの推論をおこなっていたが，実装も大幅に煩雑になる．さらに，必ずしもパラメータや返り値の型を省略することが，書きやすさに利点があるか不明瞭である．Konoha では，変数宣言の初期化だけサポートし，パラメータ型推論は今後の検討課題になっている．

3.3 影響

Konoha で導入された型推論は，初期化時に右辺値から型を推論するのみの単純な方法である．この単純な推論を導入するのみでも，言語設計に大きな影響が生じた．最後に，これらの影響をみていきたい．

†2 Principle 型などいくつか特殊な型を導入する必要がある

3.3.1 変数スコープ

動的言語は，明示的な変数宣言がないため，関数単位のスコープを採用することが多い．一方，C/C++や我々が標準語としている Java 言語の場合は，ブロック単位のスコープを採用している．スコープの違いは，両者の間に埋めがたいギャップを生み出す．

次のは，関数スコープでは問題ないが，ブロックスコープの場合となる例である．つまり，`return s;` は，スコープ外となる．

```
String typeofnum(int n) {
  if(n % 2 == 0) {
    s = "even";
  }
  else {
    s = "odd";
  }
  return s;
}
```

我々は，当初，明示的な宣言がある場合は静的言語の流儀にしたがいブロックスコープ，型推論の場合は動的言語の流儀にしたがい関数スコープと，2 種類のスコープを導入していたが，混在するスコープはそれ自体プログラマに混乱を与えることが明らかになってきた．そのため，Konoha のユーザからは動的言語らしく書けないと指摘されることが多いが，全てのローカル変数をブロックスコープに統一している．

3.3.2 総称型

Konoha は，スクリプティング言語として簡易的な Java，つまり Java1.2 を標準と考えていた．しかし，Konoha では，総称型 (generics) を導入するに至り，必ずしも簡易とは言えなくなった．総称型の導入は，型推論の導入と密接に関係している．

Konoha は，型推論を導入する以前，Array, Map などコレクションクラスは，全て Any 型の要素をもつクラスと大雑把に定義されていた．しかし，型推論の導入により，コレクションクラスの要素に対し，より厳密な推論が必要となった．

```
>>> a = [0, 1, 2]
 - [(eval):1]:(type inference) int[] a
>>> n = a[1]
 - [(eval):1]:(type inference) int n
```

Konoha では，`int[]` は `Array<Int>`の別名である．

同様に，関数オブジェクトも，静的にコールするためには型付けが必要となり，特殊な総称型である Func 型を導入している．次は，関数 `inc()` から，`int` 型をパラメータにとり，`int` 型を返す `Func<int=>int>` を推論し，関数の変数を生成している．

```
>>> int inc(int n) { return n + 1; }
>>> f = inc;
 - [(eval):1]:(type inference)
   Func<int=>int> f
>>> f(0)
1
```

Konoha は，関数は Script クラスのメソッドとして定義されている．

4 静的スクリプティング

「静的スクリプティング」とは，我々の造語である．Konoha において，スクリプト上で静的に決定された型から「プログラミングしやすさ」を向上させることを目指している．本節では，基本的なアイディアとそれを実現する拡張的な文法構造について述べる．

4.1 リソースパスによる静的な検証

Konoha は，静的言語であってもスクリプティング言語であるため，コンパイルとコード実行が同じコンテクストでおこなわれる．つまり，本来は実行時の環境に依存する制約で動的に検証すべき点であっても，コンパイラが静的に検証し，さらにコンパイル時に最適化することができる．これは，第 2.3 節において拡張されたリソースパスと組み合わせることで，威力を発揮する．

次は，リソースパスから外部リソースの存在を静的に検証し，定数に畳み込んでいる．つまり，`/etc/passwd` が存在による，条件付きコンパイルに等しくなる．

```
if(file:/etc/passwd) {
  isUnixStyle = true;
}
else {
```

```
    isUnixStyle = false;
  }
```

4.2 変換の推論

Konoha における型推論は，前節で述べたとおり，変数の初期化において右辺値からその変数の型を推論する．一方，両辺の型が決まっていたら,「型と型によって定義される」手続きを推論することができる．

もっとも単純な例は「暗示的なキャスト（変換）」と呼ばれるものである．右辺の整数値は，float 型に対して変換する手続きが挿入されている．

```
>>> float f = 1;
1.000000
```

動的言語では，変換先の型がわからないため，このような暗黙的な変換を行うことができない．しかし，データ変換は全てのプログラミングシナリオにおいて登場する処理であり，我々は，型の情報を有効活用してデータ変換をサポートすることが,「プログラミングしやすさ」の向上につながると考えている．

Konoha では，変換式を関数やメソッドから独立したモジュールとして定義し，キャスト演算子から操作することができる．また，データ翻訳の推論系 (例えば，$C \mapsto D, D \mapsto E$ なら $C \mapsto E$) を用いることで，変換式を自動的に合成する方式を導入している．これらの技術的詳細は，論文[7] において報告されている．

暗示的なデータ変換は，他方で型検査機構と衝突する．例えば，次のような場合は，プログラマが文字列から整数への変換を希望していると解釈するより，型エラーと解釈すべきである．

```
String s = "1000"
int n = s;    // 文字列から変換？
```

Konoha が標準語として Java では暗示的なキャストは upcast のみに適用し，downcast は明示的なキャスト操作を必要としていた．同じ方針を採用すると，明示的なキャスト操作を導入することになる．

Konoha では，to キーワードをつけたキャスト演算子[†3]を用い，データ変換の操作をおこなうことができる．

```
String s = "1000"
int n = (to int)s;      // やや冗長
```

キャスト演算子による明示的な変換は，自明な型宣言と同様に自明な場合もある．このような場合は，from キーワードを用いることで，自動的な変換を表明できるようになっている．接頭辞に from がつけられた式は，コンパイラによって型から変換式を推論され，適切な変換式が挿入される．

```
String s = "1000"
int n = from s;  // 変換してね
```

4.3 リソースパスの型付け

我々は，型検査を阻害しない変換操作について論じてきた．しかし，いくつかの場合では暗示的に型変換を行なっても構わない場合がある．ここでは，リソースパスの場合について論じる．

リソースパスは，第 2.3 節で定義したとおり，外部リソースを直接扱うための識別子である．リソースは，ファイルからデータベースまで，さまざまな形式が存在する，全てのリソースに共通の型として，リソースの存在を表現する Boolean 型で型付けされている．

リソースパスに対して，変換式の推論を適用すれば，各リソースに対して，適した型のオブジェクトとしてアクセスことができる．次の例は，リソースパス (URL) を InputStream としてアクセスする例である．

```
InputStream in =
    http://konoha.sourceforge.jp/;
```

我々は，リソースパスの定義に静的にどの型に変換可能か検証できる情報を付加している．そのため，コンパイル時の検査により，上記の例は，リソースの存在がチェックされ，InputStream への変換が決定される．もし変換後のオブジェクトが不変 (Immutable) であれば，定数に畳み込まれる．

より複雑な例は，次の foreach 文と組み合わせたリソースアクセスである．ここでリソースパス (URL) は，まず InputStream に変換される．さらに foreach/from によって，Iterator<String>に変換される．

```
foreach(String line from
```

†3 Java においてもキャスト演算子はデータ変換と downcast の両方に利用されている．そのため，Konoha でも to キーワードを省略しても構わない．

```
    http://konoha.sourceforge.jp/) {
  print line;
}
```

プログラマは，細かい手続きを考えなくても，リソースから求めるデータを取り出すことが可能になる．また，リソース修飾子によって，文字列として与えられたリソース，実行時までアクセスが決定できない場合であっても，同じように変換式を推論することができる．

```
url = "http://konoha.sourceforge.jp/";
foreach(String line from http:: url) {
  print line;
}
```

リソースパスは，文字列に比べ，外部リソースの識別子であるという意味論が明確である．それにより，プログラマの期待する変換を正しく推論することができる．

5 むすびに

従来，スクリプティング言語は，応用ドメインに特化したプログラムの省略や特別な演算子を多用することで，プログラムのしやすさを向上させてきた．我々は，スクリプティング言語の世界において，静的な型付けを加えたことにより，新しいスタイルのスクリプティングが可能になると考えている．型推論や型変換推論を用いることで，プログラマが書いた直観的なソースコードからプログラマの期待するコードを生成することで実現できる．Konoha では，外部リソースの識別子 (リソースパス) を導入することで，型変換推論の正しさを向上させている．本論文では，Konoha によるサンプルを示しながら，静的スクリプティングの可能性を紹介した．

謝辞 Konoha 言語は，IPA 未踏ソフトウェア創造事業「軽量オントロジレポジトリの開発」に始まり，横浜国立大学学長裁量経費による米ジョージア工科大学在外研究，総務省 SCOPE-R 若手先端 IT 研究者育成型研究開発「意味型を備えたユビキタスバーチャルマシンの開発」科学技術振興機構 JST/CREST 「実用化を目指したディペンダブル組込みオペレーティングシステム領域」等の研究補助を受けて開発を行ってきました．また，Konoha の言語設計に対して，貴重なご意見を頂いた村上直さん（KEK)，松野裕さん (産総研)，前田俊行さん（東京大学）に感謝いたします．重ねて，Konoha 言語のバグ探しに多大な貢献を頂いた倉光研究室の学生諸君に深くお礼申し上げます．

参考文献

[1] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In OOPSLA, 2009.

[2] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas Matsakis. RPython: Reconciling Dynamically and Statically Typed OO Languages. In DLS, 2007.

[3] Lars T Hansen. Evolutionary Programming and Gradual Typing in ECMAScript 4 (Tutorial), November 2007.

[4] Jeremy Siek and Walid Taha. Gradual typing for objects. In ECOOP, pages 2–27, 2007.

[5] Kimio Kuramitsu: A Map-based Integration of Ontologies into an Object-Oriented Programming Language. In *IFIP AI 2008*, pages 457–461, 2008.

[6] 倉光君郎. Konoha: ハイブリッドな型検査システムを備えたスクリプティング言語. 第 10 回プログラミングおよびプログラミング言語ワークショップ (PPL2008), 2008.

[7] 倉光君郎. セマンティックマッピングによるキャスト操作の拡張. 第 12 回プログラミングおよびプログラミング言語ワークショップ (PPL2010), 2010.

[8] 井出真広, 中田晋平, 倉光君郎. 「スクリプティング言語 Konoha によるカーネル拡張」情報処理学会システム研究会, (掲載予定) 2010.

[9] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, 2006.

[10] Roberto Ierusalimschy , Luiz Henrique de Figueiredo and Waldemar Celes The Implementation of Lua 5.0. *Journal of Universal Computer Science*. vol11. no 5. pages 1159-1176. 2005.