SML#開発最前線

うえのかつひろ

東北大学電気通信研究所

函数プログラミングの集い 2011 in Tokyo,
2011/09/17

SML#開発最前線

うえのかつひろ ← grassのひと

東北大学電気通信研究所

函数プログラミングの集い 2011 in Tokyo,
2011/09/17

SML#開発最前線

うえのかつひろ ← grassのひと

東北大学電気通信研究所

函数プログラミングの集い 2011 in Tokyo,
2011/09/17

SML#マスコットキャラ
トゲトゲ

SML#って何？

東北大学電気通信研究所で作っている（日本発の）ML系言語です.

1993年	SML# of Kansai
2003年	文科省プロジェクトe-Societyのひとつとして始動@JAIST
2006年	SML# 0.10リリース
2011年9月	SML# 0.90リリース

SML# team and collaborators

SML# Development Team:

- Atsushi Ohori (RIEC, Tohoku University)
- Katsuhiro Ueno (RIEC, Tohoku University)

in (past) collaboration with (current affiliation):

- Kiyoshi Yamatodani (Sanpu Kobo Inc.)
- Nguen Duc Huu (Hanoi University of Science and Technology)
- Liu Bochao (CNCERT, China)
- Satoshi Osaka (Advantest)

with contributions from many peoples including:

Nobuaki Yoshida, Isao Sasano, Yutaka Matsuno, Kwanghoon Choi

SML#が目指すもの

MLを
「ふつうの言語」
にすること

SML#が目指すもの

MLを
「実用的な言語」
にすること

SML#が目指すもの

輝かしき関数型言語の世界

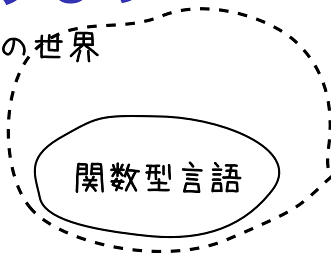
SML#が目指すもの

輝かしき関数型言語の世界

関数型言語

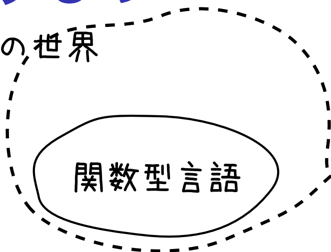
SML#が目指すもの

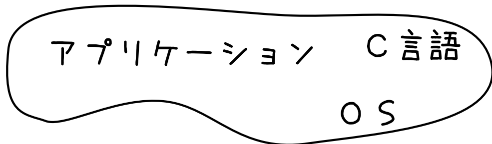
輝かしき関数型言語の世界



SML#が目指すもの

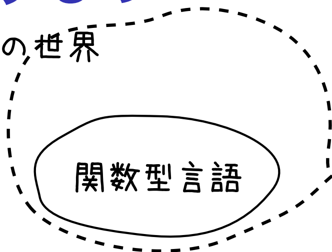
輝かしき関数型言語の世界

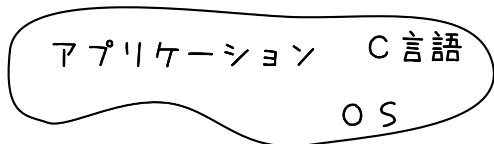




SML#が目指すもの

輝かしき関数型言語の世界

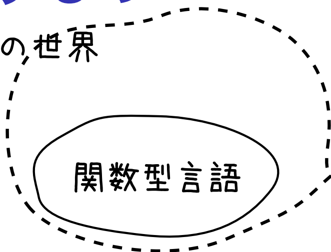


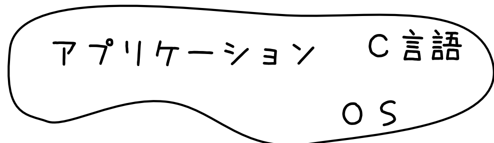


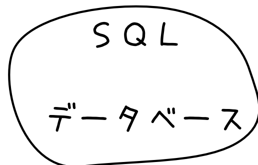
ハードウェア　ネットワーク

SML#が目指すもの

輝かしき関数型言語の世界







ハードウェア　ネットワーク

SML#が目指すもの

輝かしき関数型言語の世界

関数型言語

世界が成り立たない壁

アプリケーション

C言語

OS

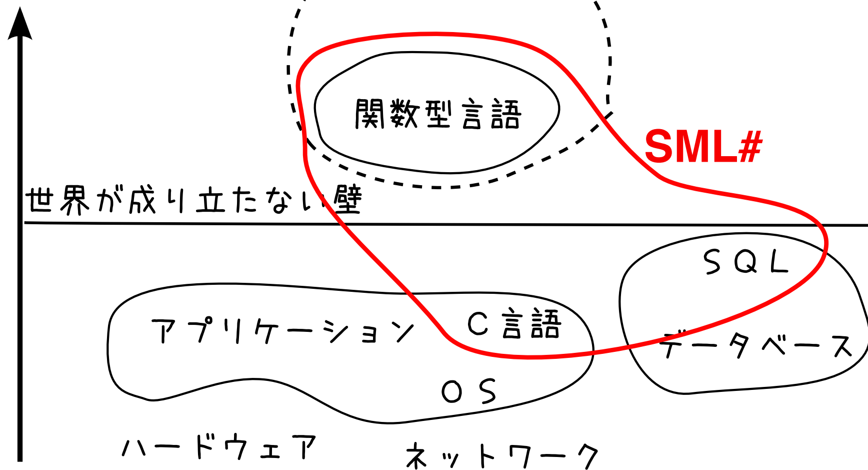
SQL

データベース

ハードウェア

ネットワーク

SML#が目指すもの



SML#が目指すもの

輝かしき関数型言語の世界

世界が成り立たない壁

関数型言語

SML#

SQL

アプリケーション

C言語

データベース

OS

ハードウェア

ネットワーク

具体的な達成項目

Standard MLとの後方互換性を失わずに

- 実用上必須の型システムの拡張
- OSやCライブラリと直接連携
- 生成するコードはネイティブ
- データ表現も全部ネイティブ
- スレッドも当然ネイティブ
- SQLの統合
- 分割コンパイル

ここ1年くらいの進捗

- SQLの統合
- non-moving GC
- pthreadサポート
- 分割コンパイル

C との連携

ただ C の関数名と型を書くだけで，C の関数を ML の関数として使える.

```
_import "関数名" : 引数の型 -> 結果の型
```

- グルーコード不要.
- 型注釈が正しければ型安全.
- データ変換や pinning などのオーバーヘッドは一切伴わない.

C との連携

puts(3) が呼べる.

```
val puts = _import "puts" : string -> int;
puts "Hello";
```

C との連携

printf(3) が呼べる. (hack)

```
val printf = _import "printf"
    : (string, int) -> int;
printf ("%8d\n", 1234);
val printf = _import "printf"
    : (string, real) -> int;
printf ("%.6f\n", 1234.0);
```

Cとの連携

C関数は第一級.

```
val puts = _import "puts" : string -> int;
map puts ["Hello", "World"];
map (fn f => f "Hello") [puts, puts];
```

C との連携

OpenGL が呼べる.

```
extern void glNormal3dv (const double *v);

val glNormal3dv = _import "glNormal3dv"
    : (real * real * real) -> unit;

val glVertex3dv = _import "glVertex3dv"
    : (real * real * real) -> unit;

app (fn (normal, vertexes) =>
        (glNormal3dv normal;
         app glVertex3dv vertexes))
    [((1.0,0.0,0.0),[(1,0,0.0,0.0), ...
```

C との連携

ML の関数を C に渡せる.

```
val glutPostRedisplay =
    _import "glutPostRedisplay" : () -> ()
val glutTimerFunc = _import "glutTimerFunc"
    : (int, int -> unit, int) -> unit
fun timer _ =
    (state := nextState (!state);
     {glutPostRedisplay ();
     glutTimerFunc (10, timer, 0));
glutTimerFunc (10, timer, 0);
```

C との連携

ML の関数を C に渡せる． (for your fun)

```
void foo
    (void(*)(void(*)(void(*)
     (void(*)(void(*)(void(*)(void)))))));
void (*(*(*bar(void))(void))(void))(void);

val foo = _import "foo"
    : ((((((()->())->())->())->())->())->()
val bar = _import "bar"
    : ()->()->()->()->()
```

C との連携

配列を更新する関数を呼べる.

```
val fft_real_radix2_transform =
    _import "gsl_fft_real_radix2_transform"
    : _import (real array, int, int) -> int

val buf = Array.tabulate (4096, waveSample)

fft_real_radix2_transform (buf, 1, 4096);
```

C との連携

fread(3) で生 double が読める． (hack)

```
val fopen = _import "fopen"
    : (string, string) -> unit ptr;
val fread = _import "fread"
    : (real array, int, int, unit ptr)
      -> int;
val fp = fopen ("doubles.bin", "r");
val buf = Array.array (10, 0.0);
val fp = fread (buf, 8, 10, fp);
```

C との連携

qsort(3) で SML#の配列をソートできる.

```
fun qsort (a, f) = _ffiapply "qsort"
    (a : 'a array,
     Array.length a : int,
     _sizeof('a),
     f : ('a ptr, 'a ptr) -> int) : unit

fun compareReal (p1, p2) =
    case Real.compare (!!p1, !!p2) of ...

qsort (realArray, compareReal);
```

C との連携

pthread_create(3) が呼べる． (experimental)

```
val pthread_create =
    _import "pthread_create"
    : (unit ptr ref, unit ptr,
       unit ptr -> unit ptr, unit ptr)
      -> int

fun threadMain _ = ...

val r = ref NULL
pthread_create (r, NULL, threadMain, NULL);
```

Cとの連携

pthread_mutex_*が呼べる． (experimental)

```
val pthread_mutex_init =
    _import "pthread_mutex_init"
    : (Word8Array.array, unit ptr) -> int

val pthread_mutex_lock =
    _import "pthread_mutex_lock"
    : ··· Word8Array.array -> int

val m = Word8Array.array
          (sizeof_pthread_mutex_t, 0w0);
pthread_mutex_init (m, NULL);
```

SQLの統合

インピーダンスミスマッチがあるなら,
統合しちゃえばいいじゃない.

- 「関係」は第一級
- 関係演算も第一級
- SQLクエリも第一級

MLの構文, 静的・動的意味論を拡張し,
MLとSQLを統合.

SQLの統合

SQLクエリそのものが書ける.

```
_sql db => select #e.name, #e.salary
           from #db.employee as e
           where #e.salary > 500;
```

SQLの統合

SQLクエリに多相型が付く.

```
val it = fn
  : ['a#{employee: 'b},
     'b#{name: 'd, salary: int}, 'c,
     'd::{int, word, char, bool, string,
          real, 'e option},
     'e::{int, word, char, bool, string,
          real}.
     ('a, 'c) _SQL.db
     -> ('d * int) _SQL.query]
```

SQLの統合

クエリを取りクエリを返す関数も自由自在.

```
fun salaryRank condFn =
  _sql db =>
    select #e.name as name,
           #e.salary as salary
    from #db.employee as e
    where condFn e
    order by #e.salary desc;
```

SQLの統合

クエリを取りクエリを返す関数も自由自在.

```
val salaryRank = fn
  : ['a#{name: 'b, salary: 'c},
     'b::{...}, 'c::{...}, 'd,
     (('a, 'd) SQL.row
      -> (bool option, 'd) SQL.value)
     -> ['g#{employee: 'a}.
         ('g, 'd) SQL.db
         -> {name: 'b, salary: 'c}
            SQL.query]]
```

SQLの統合

データベースの宣言.

```
val db = _sqlserver "dbname=sample2"
    : {employee: {name:string, age:int,
                  salary:int, deptid:int},
       department: {id:int, name:string}};
```

```
val db = "dbname=sample2"
  : {department: {id: int, name: string},
     employee: {age: int, deptid: int,
                name: string, salary: int}}
    SQL.server
```

SQLの統合

データベースと接続.

```
val c = SQL.connect db
```

```
val c = <conn>
  : {department: {id: int, name: string},
     employee: {age: int, deptid: int,
                name: string, salary: int}}
    SQL.conn
```

SQLの統合

クエリの実行.

```
val q =
  _sql db => select #e.name, #e.salary
             from #db.employee as e
             where #e.salary > 500;
_sqleval q c;
```

```
val it = <rel> : (string * int) _SQL.rel
```

SQL の統合

結果の取り出し.

```
SQL.fetchAll it;
```

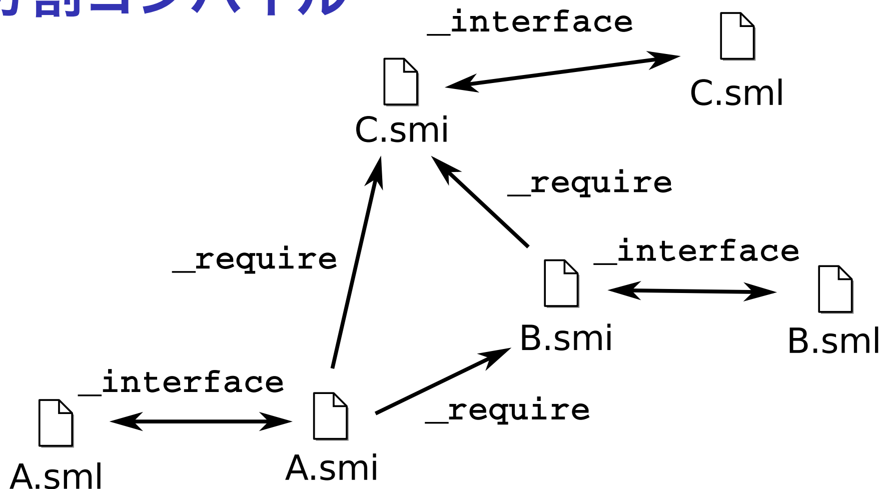
```
val it =
  [("Alice", 550), ("Bob", 700), ...]
   : (string * int) list
```

分割コンパイル

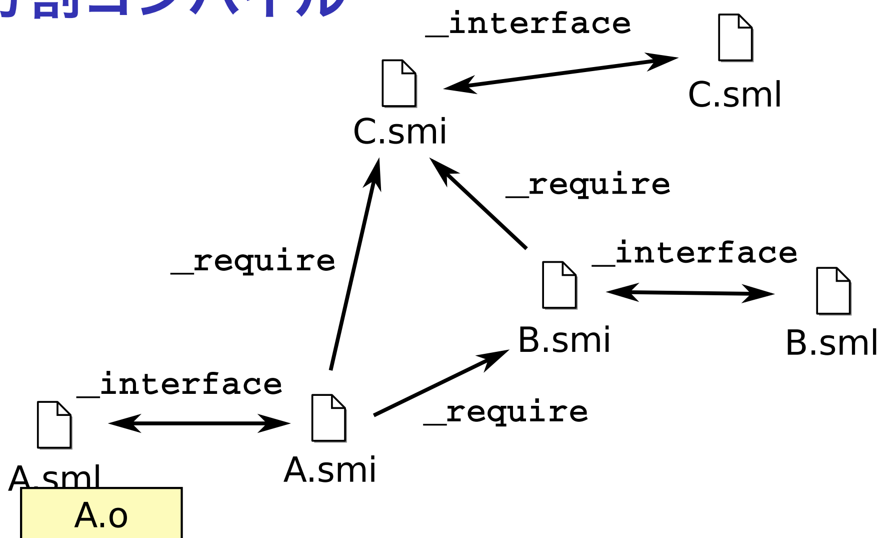
インターフェース（ヘッダー）ファイルの存在だけを要求する，真の分割コンパイル．

- ソースファイル `.sml`
- インターフェースファイル `.smi`
- コンパイルすると`.o`に．
- `ld`や`gcc`でCライブラリと一緒にリンクできる．
- functorも分割コンパイル．

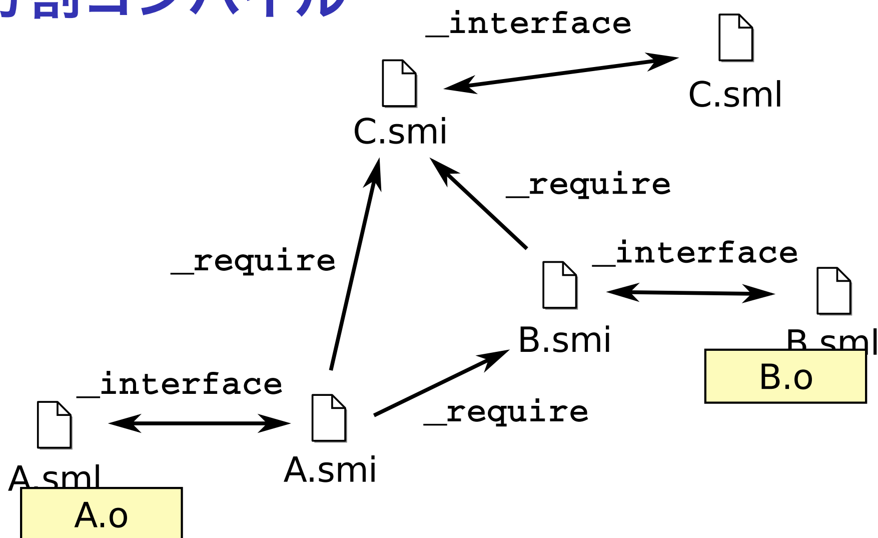
分割コンパイル



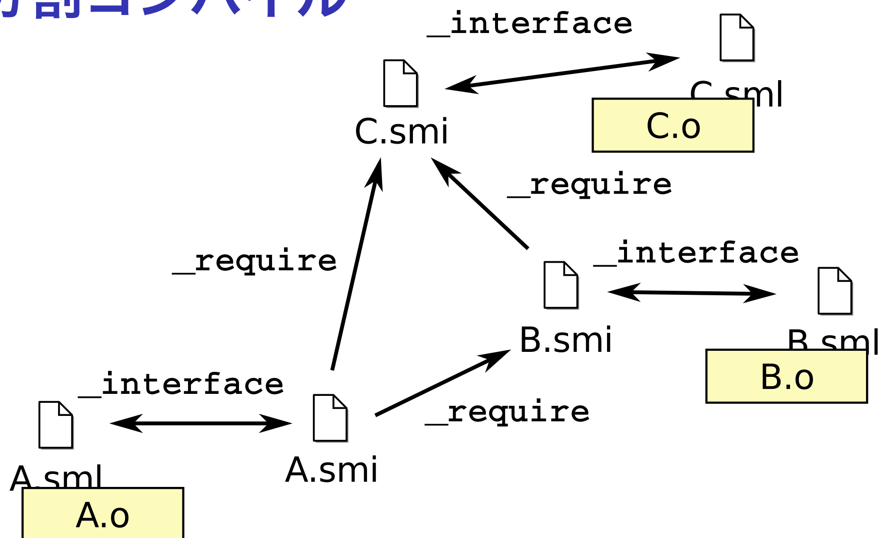
分割コンパイル



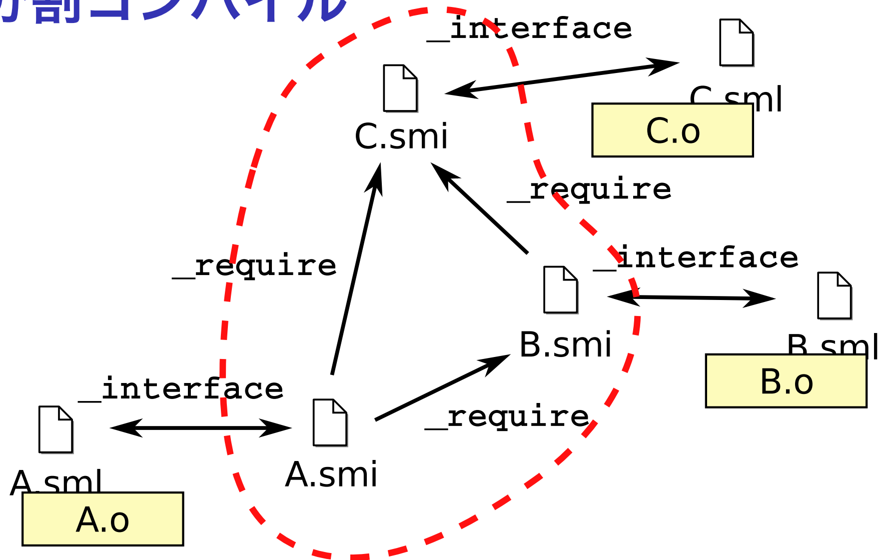
分割コンパイル



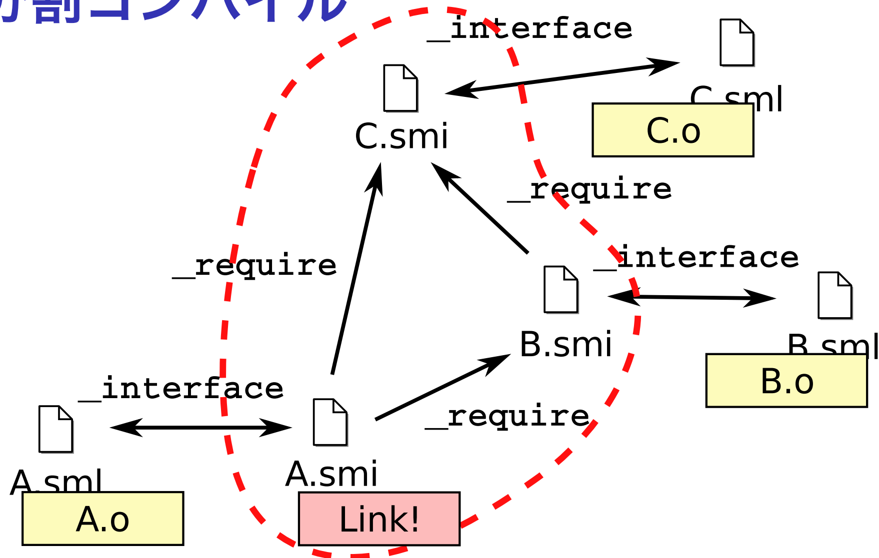
分割コンパイル



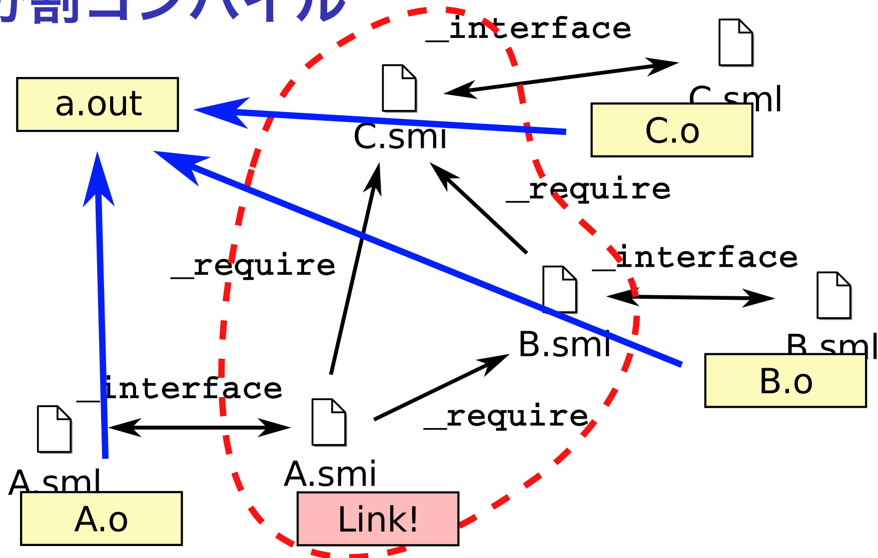
分割コンパイル



分割コンパイル



分割コンパイル



分割コンパイル

コンパイルとリンクの実行例

```
smlsharp -c A.sml
smlsharp -c B.sml
smlsharp -c C.sml
smlsharp A.smi
```

分割コンパイル

C プログラムと一緒にリンク可能

```
smlsharp openGLSample.smi -lgl -lglu -lglut

smlsharp oreApp.smi ore.o
```

分割コンパイル

コーディング例

```
B.smi:
  _require "C.smi"
  structure B =
  struct
    type t (= C.t)
    val f : t -> t
  end
```

```
B.sml:
  _interface "B.smi"
  structure B :>
  sig
    type t
    val f : t -> t
  end =
  struct
    ...
```

最後に

SML#は,

- C やデータベースとの連携を備えた
- 「ふつうの言語」を目指す

ML 系関数型言語です.

`http://www.pllab.riec.tohoku.ac.jp/smlsharp/`

または「smlsharp」や「SML」で 検索