Overview of SML# : – a new language in the ML family

Atsushi Ohori
joint work with Katsuhiro Ueno

RIEC, Tohoku University

This talk is about SML#:

Its goal is to make ML

- an ordinary practical language, and
- a practical database programming language

Some reactions from eminent language researchers would be:

- ML has been a super-advanced and highly practical language.
- There are a number of practical database programming languages.
- What's the issue?

Backgrounds

ML is indeed a great language. It is:

- Highly Productive

  You can see if you write some codes.
- Highly Reliable, and

  Polymorphic type inference is the only verification tool made into practice for programmers' daily use.
- Safe

  It is type safe and memory safe; "buffer overflow" is not even an issue.

You get all of them for free by just using ML ....

Then, everyone should have been using ML by now,...

Weaknesses of ML

Despite these advantages, ML has not yet been among the popular production languages of the industry.

There may be historical or cultural backgrounds/prejudice.

- It is a toy invented in academia...
- Functional language is difficult/inefficient/impractical...

But, more fundamentally, several important practical problems have been left unsolved.

- lack of interoperability with C
- lack of native thread support on multi-core CPUs
- lack of separate compilation and linking
- lack of database support
- improper treatment of records

Weakness (1)： improper treatment of records

Record is the most basic data constructor, which must be fully supported.

However, record operations are not quite first-class citizens in ML. In SML:

```
- fn x => map #name x;
stdIn:1.1-1.17 Error:  unresolved flex record
   (can't tell what fields there are besides #name)
```

This is actually better than some of the other systems.

A minor comment: there are several solutions. I nonetheless address this issue first, since our solution to this problem turns out to be an important step in the development of **SML#**.

Weakness (2) : lack of interoperability with C

Seamless (or at least smooth) interoperability with C is a prerequisite for a new language to be accepted in practice.

Unfortunately, in current ML, using C libraries requires:

- to understand runtime representations of ML objects,
- to write a low-level stub for data conversion, and
- perhaps to call mysterious coordination-functions for GC

which make using C not only cumbersome but also dangerous.

Weakness (3) : lack of database support

There are few real-world applications that do not use databases.

The current (ordinary) practice is string interface to SQL, where the programmer

- generates an SQL command string,
- sends it to an SQL server, and
- converts the result to appropriate data structure.

This is cumbersome, error prone and unsafe.

ML is worse: programmers cannot even use string interface to SQL.

Weakness (4) : lack of native thread support

Exploiting emerging multi/many-core CPUs is the key to future high-performance application development.

In the excising typical ML compilers, threads are implemented by their runtime system using a timer.

This implies that only one core can be used, no matter how many threads the program code generates.

Weakness (5) : lack of separate compilation

In a large software project, separate compilation and linking is essential.

Unfortunately, no ML compiler supports separate compilation and linking in the ordinary sense, i.e. the process:

- to compile a source file into an object file consisting of
  - a set of machine code blocks
  - external name definitions of the exported variables
  - external name refereces of the imported variables
- and to link an object file with
  - other ML object files independenly compiled, and
  - system libraries (libc, libm etc)

There seem to be no theoretical foundation for separately compiling Standard ML (except for typecheking.)

Sources of these weakness

Widely accepted assumptions/beliefs/folklore around ML:

- Parametric polymorphism is the principle.
- Copying collector is the choice.
- Interfaces are Types

Taking them as axioms, ML attempts to re-construct the world.

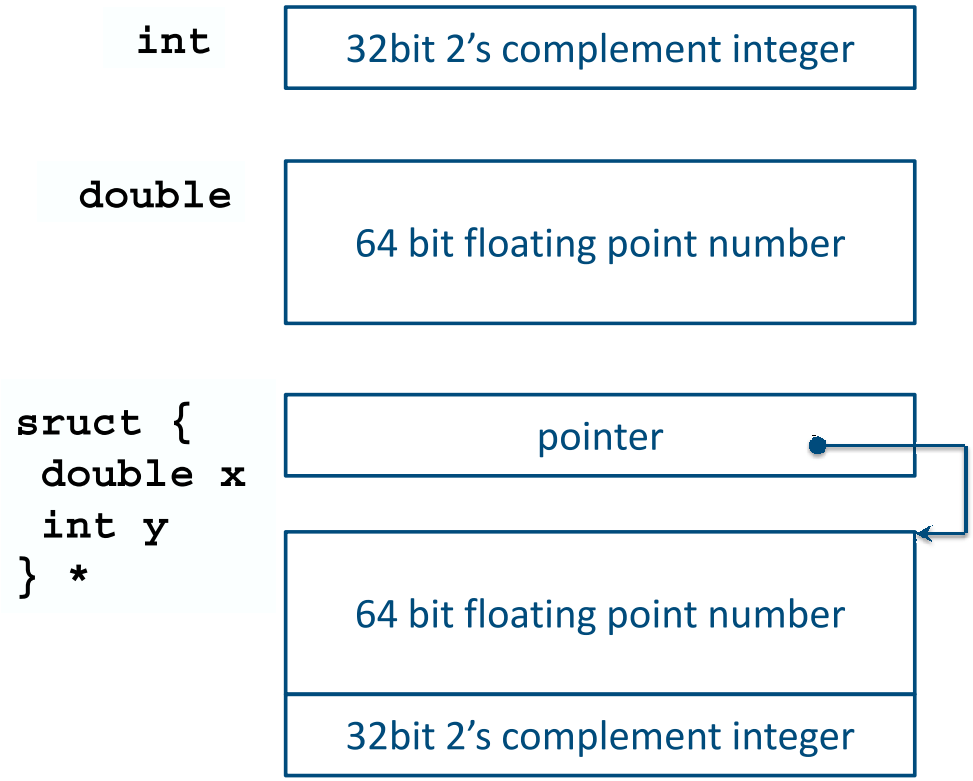
This attempt works well in theory and an idealized environment, but creates some serious problems in practice.

Let us examine some of them.

Parametric Polymorphism Principle

Since `'a` in `'a -> 'a` ranges over any types, we should only have one representation, namely 32/64 bit word (possibly with an inline tag).

In the ordinary daily computation (on any hardware), however, we have:



Some programmers would surely give up polymorphism for these data.

Copy GC Folklore

Functional programs require fast allocation (true indeed), on which folklore has it that:

> "Cheney's copying GC is the only option for primary (minor) GC".

This implies that

- there is one allocation pointer for the thread-shared global heap, and
- objects are moved at any unpredictable time.

In such an absurd environment, one reasonable strategy for the ML run-time is to act as a monitor for thread scheduling and memory allocation.

However, if we are free from the folklore, then there is a possibility of:

- having a non-moving GC with multiple allocation pointers, and
- directly using POSIX threads provided by the OS kernel.

Then you get true efficient multi-core threads for free.
This is the way it should be; OS kernels are designed to serve us.

My comments on the weakness of ML

Of course researchers have been aware that

- one-bit tag in integers causes problems
- heap-allocating floating point numbers is undesirable
- threads should be supported by the OS thread library, ···

On this situation, two comments I would like to make:

1. They are ingenious mechanisms researchers have developed to realize ML, to which my thorough respects are due.
2. Overcoming them has been shown to be quite difficult. They are the prices to pay for ML to obtain its reliability and productivity.

Unfortunately, these prices are too high to make ML an ordinary practical language.

As researchers and hackers* who love ML, we cannot stop here!

(*) A person with an enthusiasm for programming or using computers as an end in itself.

We must develop a new ML language that can be a possible choice of the industry for serious software development.

The **SML#** Project

Its goal is to develop a new language in the ML family that is:

- a proper extension of Standard ML:
  - record polymorphism (and others, e.g rank-1 polymorphism, and limited form of overloading)
  - complete downward compatibility with Standard ML
- with practically important features:
  - seamless interoperability with C
  - seamless database integration
  - separate compilation and linking
  - direct native thread support for multicore CPUs

SML# Development Team (past; current affiliation):

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

in (past) collaboration with (past; current affiliation):

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

Precursors of **SML#**

- Machiavelli, 1987 – 1989, University of Pennsylvania
- Standard ML# of Kansai (SML#), 1992 – 1993, Oki Electric
- SML extension, 1995 – 1997, Kyoto University
- **SML#** Project, 2003 – 2005, JAIST; 2005 – present Tohoku University

Practicality is not an easy object to obtain.

SML# feature: record polymorphism I

Modular programming with records:

- organize the system as a set of modules, each of which operates on specific attributes represented as record fields, and
- compose them through record polymorphism.

Example: points (the OO "benchmark" in '80s — '90s).

1. design a module on X coordinate

```
fun moveX (p as {X,Vx,...}) = p # {X = X + Vx};
fun accelerateX (p as {Vx,...}) = p # {Vx = Vx + 0.0};
fun updateX p = (accelerateX o moveX) p
```

```
val updateX : ['a#{Vx:real,X:real}. 'a -> 'a]
```

2. similarly on Y coordinate

```
val updateX : ['a#{Vx:real,X:real}. 'a -> 'a]
```

SML# feature: record polymorphism II

3. compose them via record polymorphism

```
fun update p = (updateX o updateY) p
```

```
val update : ['a#{Vx:real,Vy:real,X:real,Y:real}.'a -> 'a]
```

More generally,

```
fun composit p =
  let
    val newP = process1 {newP=p,oldP=p}
    val newP = process2 {newP=newP,oldP=p}
    ....
  in
    newP
  end
```

This scales up to complicated systems such as game programs.

SML# feature: seamless database integration

In **SML#**,

- SQL expressions are first-class citizens in the language, and
- they are evaluated by a database server.

An SQL command, e.g.

```
Q = SELECT name, age
    FROM people
    WHERE age <= 25
```

which is a shorthand for the following verbose syntax:

```
Q = SELECT person.name AS name, person.age AS age
    FROM people AS person
    WHERE (person.age <= 25)
```

can be directly written as an **SML#** expression.

SML# feature: seamless database integration

```
# val Q = _sql db => select #person.name as name,
                             #person.age as age
                     from #db.people as person
                     where SQL.<= (#person.age, 25);
val Q = fn
     : ['a#{people:'b},
        'b#{age:int, name:'d},
        'd::{int, word, char, string, real, 'e option},
        'e::{int, word, char, bool, string, real}.
        'a SQL.db -> {age: int, name: 'd} SQL.query]
```

Inferring a principal type ensures seamless integration. Remember: types solely governs program composition.

In addition, you get a lot of type errors (in fact, all of them), of course.

Technical Development of SML#

SML# is realized by combining the following

1. Polymorphic type system for databases
2. Record polymorphism and type-directed compilation
3. Type-directed record compilation
4. Natural data representation
5. Separate compilation and linking
6. Efficient non-moving and fully concurrent (on-the-fly) GC

The key technique: type-directed record compilation

The key insight underlying the development of **SML#** compiler is

> *"Polymorphism" is just an abstraction, and not a computational reality. So it must be compiled out.*

The first example is the polymorphic record calculus and its compilation (Ohori 1992, 1995).

Consider the term:

```
fn x => #name x
```

This is apparently polymorphic, but it is also apparent that this must be compiled out to efficient load instruction.

Idea of record compilation

It is based on the following ideas:

1. Information needed to compile a polymorphic function is abstracted at type abstraction and applied at type application.
2. The needed information is encoded as a type that denotes the unique needed value. In the example, `index(`τ`, name)` denotes the index value of the `name` field of a record type τ.

This mechanism is far-reaching in compiling out various aspects of polymorphism, which we have used to develop **SML#** throughout, including

- natural data representation
- first-class overloaded primitives
- separately compiling functors
- colling convention compilation for LLVM

SML# for ordinary practical software development

Some joint projects:

- Advanced ERP system with NEC Software Tohoku.
  - Retrieving the current state of various RDB databases
  - Real-time computation of costs, overhead, employee loading etc
  - Simulation of future performance
  - Visualization
- Programming framework for highly available storage infrastructure with Hitachi Solutions Higashinihon
- Server framework for Hybridcast (a new generation TV) with NHK

Projects being planned with software companies:

- Medical big data analysis
- Software development platform for embedded systems

The Current Status of SML#

The current release: version 1.20 supporting

- record compilation,
- database integration,
- native multi-threads

available from

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

The following will soon be available in the forthcoming **SML#** 2.00.

- concurrent GC for native threads on multi-core CPUs
- an LLVM back-end

Future Perspective

ML (**SML#**) is :

- Highly Productive
- Highly Reliable
- Safe, and
- Higly Interoperable.

We also note that ML is

- a declarative language of expressions, with
- a static type system

which make an ideal basis for implementing specifications verified by CafeOBJ.

So we have a long term future plan: integrating CafeOBJ with **SML#**.

将来展望：高信頼ソフトウェア工学基盤 (Long-term speculative plan)

背景

- 我が国のソフトウエア生産における競争力の相対的な低迷.
- 次世代産業におけるソフトウェアの重要性

目的：ソフトウエア学術研究の活性化と我が国のソフトウェア産業の新生

1. 高信頼言語と代数仕様記述言語を統合する新たな枠組みの構築
2. 同枠組みを基礎とした高信頼ソフトウェア生産基盤の開発と普及による我が国の高度ソフトウェア生産の生産性と信頼性の飛躍的向上

戦略：高信頼言語 SML#と代数的仕様記述言語 CafeOBJ との統合

- SML#ご紹介の通り.
- CafeOBJ(JAIST 二木グループ)：世界を代表する代数的仕様記述言語
  - 適切な抽象度での仕様の記述能力
  - 堅牢な基礎と仕様の検証実行可能な処理系

長年の蓄積を持つ我が国発の両技術を統合し，目的達成をめざしたい.

高信頼ソフトウェア工学基盤のイメージ

1. 超高機能高信頼プログラミング言語
   - 組込ソフト開発を可能にする OS との連携・統合
   - ビッグデータ解析等を可能にするデータ処理能力
   - メニーコア上の超並列処理
2. 代数仕様記述言語に高信頼言語を統合したソフトウエア生産基盤
   - 要求仕様からコードに至る，種々の妥当な抽象度での，システム仕様の検証
   - 仕様定義・検証とコード開発を系統的に行うプログラミング環境
3. 検証済みソフトウエアレポジトリクラウドの構築
   - 同基盤を用いたライブラリ群の仕様記述・検証
   - 検証済みライブラリのクラウドを通じた提供
4. 教育カリキュラムの開発と高度ソフトウェア開発人材育成プログラム
   - 高信頼言語活用技術，代数仕様記述検証技術
   - 高度ソフトウェアアーキテクト人材養成