夏のプログラミング･シンポジウム「ビューティフル･インターフェイス」2014.8.24

ATS言語を使って不変条件をAPIに強制する

岡部 究[1,a)]

概要：「モノのインターネット」という概念が提唱されています。バスケットボールのような、なんの変哲もないモノがインターネットに接続されるようになるでしょう。そのような機器は複雑な機能を持つ一方、現代の組み込み機器はC言語を用いて開発されており、不具合を誘発する危険性がつきまとっています。本論文ではこの問題に対する1つの解決策としてATS言語を紹介します。このATS言語はMLのような関数型言語で、静的な強い型を持ちます。また依存型を持ち、証明器としても機能します。さらにこの言語の線形型を使うことで、メモリやロックのようなリソースの制御を安全に行なうことができます。ATSコンパイラはC言語を経由して実行バイナリを生成するため、強い型による不変条件の強制は常にソースコードと同期しています。最後にわずか8kBのメモリしか持たないArduinoボード上でATS言語を使って設計したアプリケーションを動作させ、組み込み領域にATS言語を適用可能であることを示します。

キーワード：ATS, 関数型言語, 依存型, 線形型, 組込開発

1. はじめに

「モノのインターネット (IoT：Internet of Things)」という概念が提唱されています。バスケットボールのような、なんの変哲もないモノがマイクロチップを内蔵してインターネットに接続されるような世界を予想した概念がIoTです。そのIoTによる製造業への経済効果は2850億ドルと試算されています (ガートナー調べ [14])。このようなIoTデバイスに対する要件は以下のようなものになるでしょう。

- インターネットに接続可能なネットワークプロトコルを内蔵
- 短時間/少工数による設計
- 個人情報を保管/送信する機能
- セキュリティへの配慮
- ネットワーク非対応な機器よりもさらにインテリジェントな機能
- 安価な価格を実現するため非力なハードウェアの使用

一方、現代の組み込み機器はC言語やC++言語を用いて慎重に開発されています。これはこれらの言語を使った開発手法がバッファオーバーランのような様々な不具合を誘発するためです。このような不具合の誘発を防ぐために、VDM [1] やZ [9] のような形式手法が用いられます。しかし、これらの形式手法はモデルベースのものであり、実際に動作するC言語やC++言語で記述された設計と同期していません。これらのモデルとコードによる設計の同期には、やはり人間の力が必要になります。つまり製品のバージョンを更新する毎にこの形式手法モデルとソースコードを同期さ

[1] METASEPI DESIGN
[a)] kiwamu@debian.or.jp

せる工数が恒常的に発生してしまいます。

この慎重な開発はPOSIX API上でのアプリケーション開発と比較にならないほどの工数を必要とします。にもかかわず、IoTデバイスはそのような開発者にPOSIX上のアプリケーションレベルの機能を要求します。このC言語を中心とした開発プロセスは今後持続可能なものなのでしょうか？ソフトウェア科学/工学はこの問題に対して、有効な手を打てないのでしょうか？

本論文ではこの問題に対する1つの解決策としてATS言語 [10] を紹介します。このATS言語はMLのような関数型言語で、静的な強い型を持ちます。またこの言語は依存型を持ち、Coq [4] のような証明器としても機能します。さらにこの言語の線形型を使うことで、メモリやロックのようなリソースの制御を安全に行なうことができます。ATS言語はC言語を経由して実行バイナリを生成するため、強い型による不変条件の強制は常にソースコードと同期しています。そのため製品のバージョンを数年にわたって更新する場合でも型とソースコードは機械的に同期されます。最後にわずか8kBのメモリしか持たないArduino Mega 2560ボード [2] 上でATS言語を使って設計したアプリケーションを動作させ、組み込み領域にATS言語を適用可能であることを示します。

2. ATS言語とは

ATS言語は、依存型を持つ関数型言語であるDependent ML [13] の後継にあたり、ボストン大学のHongwei Xiによって開発されています。

ATS言語のプログラムは大きく3つの要素から成り立っています (図1)。 動的な世界 (Dynamics)、静的な世界 (Statics)、証明の世界 (Proofs) です。 動的な世界は通常のプログラミング言語と同じ実際に実行されるロジックです。動的な世界はATSコンパイラによってC言語に変換され、GCCによって実行バイナリになります。 静的な世界はHaskellやOCamlのような型推論を持つ言語における型のことです。 静的な世界は実行時エラーを防ぐ目的で、型の力で動的な世界を制約します。証明の世界は依存型と線形型に分類できます。 依存型はCoqのような証明器として使うことができ、動的な世界のロジックに証明を与えることができます。 線形型はリソースを取り扱うことができ、例えばGCに頼らない動的な世界によるメモリ領域の管理を安全に行なうことができます。 動的な世界はコンパイルされて実行バイナリになりましたが、静的な世界と証明の世界はATSによるコンパイル時に評価 (型検査) され、実行バイナリにはなりません。 しかし、静的な世界と証明の世界の検査に失敗すると、動的な世界のコンパイルも中止されます。

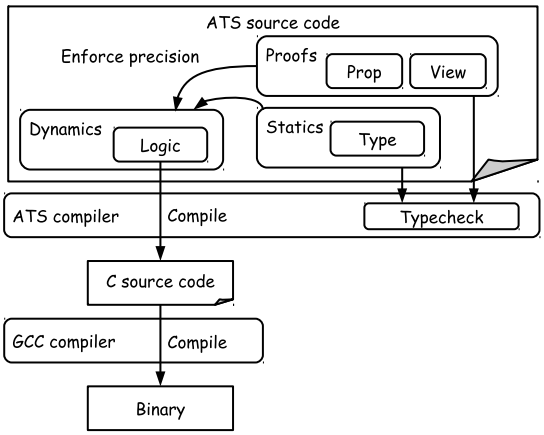

図 1 ATS言語の構成要素とコンパイルフロー

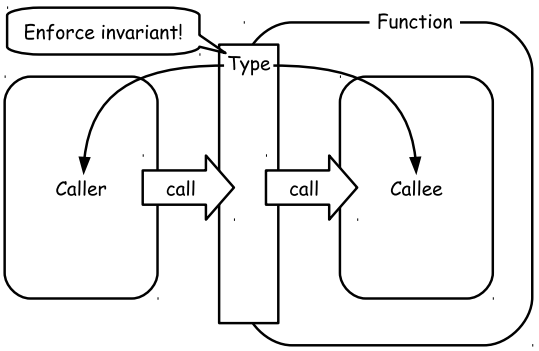

図 2 型による制約の強制

ATS言語ではインターフェイス宣言に依存型と線形型による強い制約をつけることができます (図2)。通常の言語ではこのような制約はAPIの呼び出し元にのみ強制されますが、ATS言語では型推

論によって、関数の呼び出し元と呼び出し先の双方に型による制約を与えることができます。

3. ATS の型について

ATS 言語の型は依存型と線形型をそなえています。しかし、実行可能な言語においてこれらの強力な型を用いるイメージが想像しにくいかもしれません。そこで、本章では線形型を使ったリスト (線形リスト) を例にして、ATS 言語の型の実例を解説します。

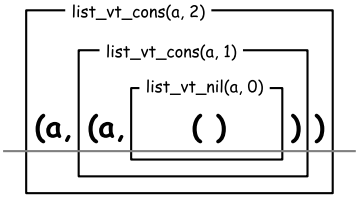

図 3 線形リストの型と構造

図 3 に線形リストの型と構造を図示します。線形リストの動的なデータ構造は LISP のような通常のリストそのものです。しかし静的な型表現がこの動的なデータ構造に強制されています。ATS 言語の基盤ライブラリである prelude では以下のように線形リストの型を宣言しています。

```
(* ファイル: prelude/SATS/list_vt.sats *)
datavtype
list_vt0ype_int_vtype (a:vt@ype+, int) =
  | {n:int | n >= 0}
    list_vt_cons (a, n+1) of
      (a, list_vt0ype_int_vtype (a, n))
  | list_vt_nil (a, 0) of ()
stadef list_vt = list_vt0ype_int_vtype
vtypedef
List_vt (a:vt0p) = [n:int] list_vt (a, n)
```

先の LISP のようなリスト構造はリストの長さ情報を持っていません。ところが、この線形リストは静的な型 (依存型) によって長さを保持しています。$\{n : int | n >= 0\}$ の宣言は全称量化を使って静的な変数 n を導入しています。この n は 0 以上であるべきであるという制約も同時に宣言されています。さらに線形リストの終端ではこの静的な値は 0 に固定されています。すなわちこの静的な値 n は線形リストの長さを表現していることになります。この静的な値はコンパイル時まで存在していて、実機で動作する実行バイナリには含まれないことに注意してください。ATS 言語はどのような動的な型表現も持ちません。静的な型表現のみを扱います。

datavtype 宣言は線形型のデータ宣言です。そのため、このリスト構造は生産と消費という概念を持っています。生産しないかぎり、そのデータ構造をさわることはできません。また生産されたデータ構造は静的な文脈が終了するまでの間に消費されなければなりません。これらの制約をプログラムが破った場合にはコンパイル時エラーとなります。

この線形リストを使った関数群をいくつか見てみましょう。list_vt_make_pair 関数 (図 4) は 2 つの要素 x1 と x2 を引数に取り、線形リストの型 list_vt を返します。この関数は list_vt 型を生産することが型宣言からわかります。この list_vt 型はこれ以降のコードのどこかで消費しなければなりません。消費し忘れた場合にはコンパイル時エラーとなります。

```
fun{x:vt0p}
list_vt_make_pair (x1: x, x2: x):
                        <!wrt> list_vt (x, 2)
```

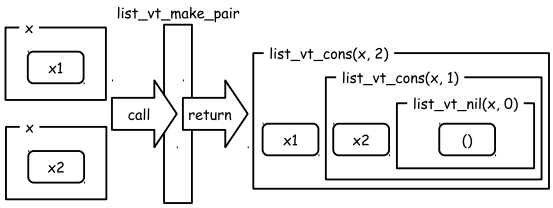

図 4 2 つの要素をペアにして線形リストを作る

では生成された線形リストはどのように消費 (すなわちそのメモリ領域を解放) すれば良いのでしょうか？それは単に free 関数を呼び出すだけで

す。この free 関数は多種の関数実体によってオーバーロードされており、線形リストの場合には list_vt_free 関数 (図 5) が呼び出されます。ソースコードに付けられた型からも List_vt 型を受けとり void 型を返す、つまり List_vt 型を消費していることがわかります。この List_vt 型が保持していたはずのメモリリソースを安全に解放することは ATS 言語の動的な部分の実装の責務です。つまりこのインターフェイスでは「線形リスト型の消費」は線形型による静的な検査の対象であり、「線形リストが保持していたメモリ領域の解放」は動的なコードの責務です。プログラマはこの 2 つが同じ意味になるように設計を行なうことで、動的なコードの挙動の中に静的な意味論を見出すことが可能になります。動的な意味論をコンパイル時に扱うことは多くの困難がありますが、静的な意味論はコンパイル時の型検査によってコンパイル時エラーとして扱うことができるのです。すなわち、ここでの「静的な意味論」とはまさに動的なプログラムの持つ「不変条件」に他なりません。

```
fun{x:t0p}
list_vt_free (xs: List_vt (INV(x))):
                              <!wrt> void
overload free with list_vt_free
```

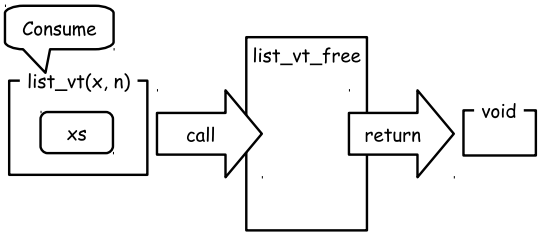

図 5 線形リストを解放する

次に線形リストを append する関数である list_vt_append 関数 (図 6) を見てみましょう。この関数は 2 つの線形リスト xs1 と xs2 を取り、1 つの append された線形リストを返します。この 2 つの線形リスト xs1 と xs2 は消費されます。そして append された線形リストが生産されます。以下のソースコードで append された線形リストに $n1 + n2$ という依存型の制約がついていることがわかります。$\{n1, n2 : int\}$ という静的な変数が全称量化によって導入されています。この 2 つの変数ははそれぞれ xs1 と xs2 の線形リストの長さを表わしています。つまり append された線形リストの長さが $n1 + n2$ であることを依存型を使って制約しています。この型は list_vt_append 関数の呼び出し元にも制約を強制しますが、呼び出し先である list_vt_append 関数の実装にも制約を強制します。この制約をプログラマが破った場合にはコンパイル時エラーとなります。

```
fun{
a:vt0p
} list_vt_append
  {n1,n2:int} (
  xs1: list_vt (INV(a), n1),
  xs2: list_vt (a, n2)
) :<!wrt> list_vt (a, n1+n2)
```

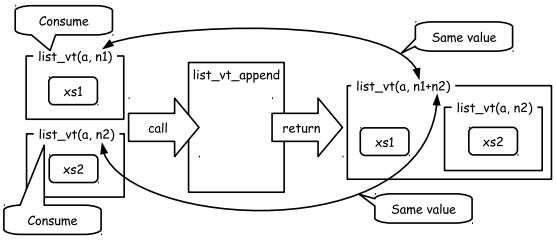

図 6 線形リストの append

また、線形リストは [] を使って添字指定で要素を読み出すこともできます。この [] もまたオーバーロードされた関数で、その関数実体は渡される型によって異なります。線形リストの場合にはその実体は list_vt_get_at 関数 (図 7) です。この関数にも全称量化 $\{n : int\}$ が導入されており、それは添字アクセスされる線形リストの長さを表わし、添字アクセスするインデックスがその線形リストの長さより小さくなければならないことが依存型で宣言されています。すなわち線形リストの範囲外を添字アクセスしてしまう不具合をコンパイル

時に予防できます。さらに線形リストの引数 xs の型には！マークが付記されています。このマークは続く線形型を消費しないことを表わします。つまり添字指定で線形リストの要素を読み出しても、その線形リストは消費されません。これは納得できるインターフェイスでしょう。またこの関数の利用者は関数内部で線形リストが使用していたメモリが解放されないことを信じることができます。

```
fun{x:t0p}
list_vt_get_at{n:int}
  (xs: !list_vt (INV(x), n),
   i: natLt n):<> x
overload [] with list_vt_get_at
```

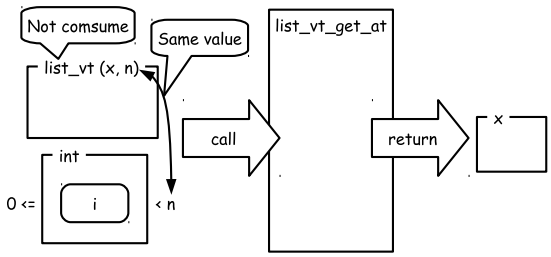

図 7 線形リストの添字指定

最後に線形リストの reverse である list_vt_reverse 関数 (図 8) を見てみましょう。この関数は、引数である xs を消費し、逆順にした新しい線形リストを生成します。全称量化による制約は「引数と返り値の線形リストの長さが同じ」ということだけ強制しています。そのためこの型では「線形リストの中身が本当に逆順になっているのか？」ということまでは強制できていないことがわかります。それでも、引数として渡された逆順になる前の線形リストは消費されているので、仮に list_vt_reverse 関数の実装がその線形リストのメモリ領域を逆順にした新しい線形リストのために再利用したとしても、元の線形リストは消費されているためにプログラマが誤って元の線形リストにアクセスする危険性を防止できます。

```
fun{x:vt0p}
list_vt_reverse{n:int}
  (xs: list_vt (INV(x), n)):
            <!wrt> list_vt (x, n)
```

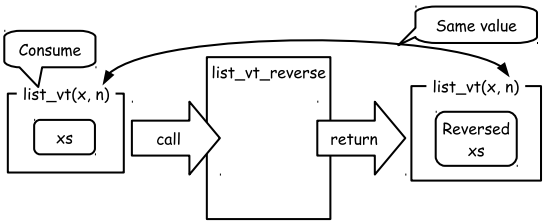

図 8 線形リストの reverse

4. Arduino における ATS プログラミング

ATS 言語による組み込み開発の適正を評価するために、Arduino Mega 2560 ボード上で動作するアプリケーションを ATS 言語で試作しました。このボードは以下のような仕様です。

Architecture 8-bit Harvard architecture
Microcontroller ATmega2560
Flash Memory 256kB
SRAM 8kB
Clock Speed 16MHz

本稿を執筆している段階では、さらにメモリの少ない Arduino Uno ボード [3] 上でも同様のアプリケーションの動作に成功しています。このアプリケーションのソースコードは [6] から入手可能です。

本アプリケーションのソフトウェアアーキティクチャは図 9 のようになります。本アプリケーションはハードウェアの機能として GPIO とシリアルポートを使用します。GPIO は Arduino の GPIO ライブラリを経由して ATS 言語から使用します。シリアルポートのレジスタ群には ATS 言語から直接さわります。ただし、シリアルポートから読み書きするデータを一時保管するリングバッファについては Arduino ライブラリを流用することにし

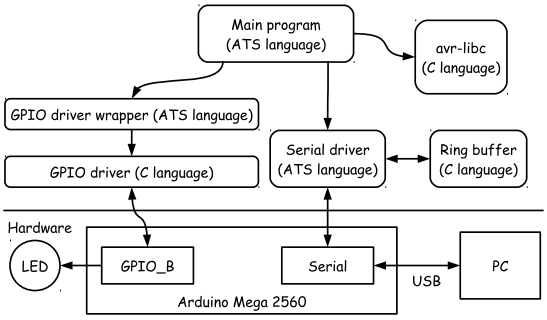

図 9　Arduino 上における ATS アプリケーション例

ました。この 2 つの抽象化によって、どのような OS のサポートなしに、さらに GC や malloc ヒープさえ準備せずに ATS 言語を使ってアプリケーションを書くことができます。

なぜ GC と malloc ヒープなしに強い型を用いた関数型プログラミングを行なうことが可能なのでしょうか？それは ATS 言語が値渡し (call-by-value) をコアにしていることと、線形型を使ったメモリリソースの管理によって実現されています。線形型を使うことで固定領域のメモリを安全に使用することが可能になり、さらにその領域外へのアクセスをメモリ領域のサイズに依存した依存型を使って防止することも可能です。

5. ATS による組込開発手法の考察

前章で ATS 言語を使って OS 不在の領域における組み込み開発が可能なことを示しました。本章では ATS 言語を用いた組み込み開発の利点と欠点についてまとめます。

その利点は静的な強い型と依存型、線形型を OS なしに利用することができることです。これまでこのような型による制約は POSIX API のようなインターフェイスを下支えする OS なしには利用できませんでした。そのため組み込みにおける開発では、形式手法を用いた後付けによる制約強制の手法が盛んに用いられてきました。ATS 言語を使えば、型による制約によって実機で動作させるソースコードに制約を付けることが可能になります。製品バージョンを重ねてソースコードが成長したとしても、この制約は機械的に引き継がれることになります。これは今までの形式手法とは全く異なるレベルの画期的な技術です。

一方、欠点もあります。GC を持つ Haskell や OCaml のような関数型言語と比較すると、GC を使わずに線形型を用いる ATS プログラミングは他のプログラミング言語には見られない特殊な設計能力が要求されます。近年、線形型を使用可能な言語として Rust 言語 [5] も人気ですが、Rust 言語は純粋な線形型をではなくポインタに限定して線形型を用います。著者も Haskell と OCaml のプログラミング経験はありましたが、ATS 言語の依存型と線形型に慣れるのにはかなりの時間が必要でした。現在組み込み開発の現場では C 言語と C++言語を使っているため、一般的な関数型言語とさらに加えて ATS 言語を習熟するのは容易ではないでしょう。

6. 結論と今後の課題

ATS 言語による強い型の強制が OS さえないような組み込みドメインに対して有用であることを述べました。しかし、ATS 言語の力を普及させるためには ATS 言語の教育プログラムが重要であることも考察しました。そこで筆者は 2 つのプロジェクトを推進しています。

1 つ目は、Japan ATS User Group (JATS-UG) [8] です。このプロジェクトではインターネット上で入手可能な ATS 言語に関する文献を日本語に随時翻訳しています。特に「ATS プログラミング入門 [11]」は有用でしょう。本稿執筆時点ではこのドキュメントが最も体系的に ATS 言語を解説しています。また、線形型に関しても本稿執筆時点では日本語で読める最も詳しいドキュメントでしょう。また ATS 言語の型システムを理論的に解説した論文 “Applied Type System (Extended Abstract)” [12] も日本語訳しています。詳細は先の JATS-UG の Web ページをご覧ください。

2 つ目は、Functional IoT [7] です。このプロジェクトでは 8-bit AVR と 16-bit MSP430、32-bit ARM Cortex-M シリーズの 3 つのアーキティクチャを対象として、ATS 言語のような強い静的な型を持つ言語を使ったマイコンプログラミングを

試行しています。このプロジェクトにはさらに 2 つの目的があります。1 つ目はこのプロジェクトを通して ATS 言語のような組み込み領域に適用可能なプログラミング言語をサーベイすること。2 つ目は製品よりも小さな領域の関数型組み込みプログラミングを通じて、そのデザインパターンを蓄積することです。日本のみならず世界を見わたしてもこの規模のアーキティクチャ上で動作する関数型プログラミングの実例はほとんどありません。そのため依存型や線形型をどのように駆使すべきであるのか、そのイディオムが圧倒的に不足しています。

最後に、先の論文 “Applied Type System” は 2004 年に発行されています。しかし ATS 言語の作者である Hongwei Xi の知るかぎりでは、この型システムの実用例は ATS 言語のみであるそうです。関数型言語の持つ安全性が OS の外の領域でも活用できるために、この Applied Type System のさらに先にある理論の探求がさかんに行なわれることを願ってやみません。

謝辞　ATS 言語に関して助言をくれたボストン大学の准教授である Hongwei Xi と ATS コミュニティに感謝します。

質疑・応答

A　線形型に val で別名を作るとどうなるのですか？

岡部　消費されます。以下に例を付記します。

```
(* コンパイル NG: let val でも線形型が消費されてしまう *)
#include "share/atspre_staload.hats"
implement main0 () = {
  val l1 = list_vt_make_pair<int> (1, 2)
  val l2 = l1
  val () = let val l3 = l2
           in println! l3 end
  val () = free l2
}

(* コンパイル OK: val で線形型が消費される *)
#include "share/atspre_staload.hats"
implement main0 () = {
  val l1 = list_vt_make_pair<int> (1, 2)
  val l2 = l1
  val () = println! l2
  val () = free l2
}
```

B　リングバッファを ATS で書くのは難しいのですか？

岡部　公式ドキュメントに例があるぐらいなので、リングバッファ自体は簡単です。しかし、スレッドセーフ化や再入可能にするのはそれなりに難しいと思います。

C　ATS 言語でマルチスレッドを生かしたプログラミングをするにはどうすれば良いのですか？

岡部　セッションのようなものを作ることになります。例えば mutex のロックと開放では、その間にセッションが存在すると考えることができます。セッションの間は静的な型を効果的に使うことができます。

D　ATS に辿りつくまでの Metasepi プロジェクトの歴史について簡単に説明してください。

岡部　ATS を使ったイテレーションは 2 番目です。1 番目では Haskell 言語と jhc コンパイラを使っていました。Haskell 言語の欠点は、メモリ領域の扱いがルーズであることと、マシン表現と言語表現にギャップがあることです。一方 ATS 言語の欠点は、抽象化の機能が Haskell 言語と比較して弱いことです。

参考文献

[1] Agerholm, S. and Larsen, P. G.: The IFAD VDM Tools: Lightweight Formal Methods., *FM-Trends* (Hutter, D., Stephan, W., Traverso, P. and Ullmann, M., eds.), Lecture Notes in Computer Science, Vol. 1641, Springer, pp. 326–329 (1998).

[2] Arduino: Arduino Mega 2560, (online), available from ⟨http://arduino.cc/en/Main/arduinoBoardMega2560⟩ (accessed 2014-10-31).

[3] Arduino: Arduino Uno, (online), available from ⟨http://arduino.cc/en/Main/ArduinoBoardUno⟩ (accessed 2014-10-31).

[4] development team, T. C.: The Coq proof assistant reference manual, LogiCal Project (online), available from ⟨http://coq.inria.fr⟩ (accessed 2014-10-31).

[5] Hoare, G.: The Rust Programming Language, Rust Project (online), available from ⟨http://www.rust-lang.org/⟩ (accessed 2014-10-31).

[6] Okabe, K.: ATS programing on Arduino, Metasepi project (online), available from ⟨https://github.com/fpiot/arduino-ats⟩ (accessed 2014-10-31).

[7] Okabe, K.: Functional IoT, Metasepi project (online), available from ⟨http://fpiot.metasepi.org/⟩ (accessed 2014-10-31).

[8] Okabe, K.: Japan ATS User Group, Metasepi project (online), available from ⟨http://jats-ug.metasepi.org/⟩ (accessed 2014-10-31).

[9] Spivey, J. M.: *The Z Notation: A Reference Manual*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989).

[10] Xi, H.: The ATS Programming Language, Boston University (online), available from ⟨http://www.ats-lang.org/⟩ (accessed 2013-11-18).

[11] Xi, H.: ATS プログラミング入門，Metasepi project（オンライン），入手先 ⟨http://jats-ug.metasepi.org/doc/ATS2/INT2PROGINATS/index.html⟩（参照 2014-10-31）.

[12] Xi, H.: Applied Type System (extended abstract), *post-workshop Proceedings of TYPES 2003*, Springer-Verlag LNCS 3085, pp. 394–408 (2004).

[13] Xi, H. and Pfenning, F.: Dependent Types in Practical Programming, *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, New York, NY, USA, ACM, pp. 214–227 (online), DOI: 10.1145/292540.292560 (1999).

[14] 三島一孝：製造 IT ニュース：「モノのインターネット」製造業への経済効果は 2850 億ドル――ガートナー- MONOist，アイティメディア株式会社（オンライン），入手先 ⟨http://monoist.atmarkit.co.jp/mn/articles/1310/15/news009.html⟩（参照 2014-10-31）.