情報処理学会論文誌　プログラミング　Vol. 1　No. 2　85–99 (Sep. 2008)

タプル空間によるブラウザ間通信を備えたScheme処理系の開発

原　　　悠[†1,∗1]　鵜　川　始　陽[†1,∗2]
湯　淺　太　一[†1]　八　杉　昌　宏[†1]

JavaScript による高機能な Web アプリケーションの開発に不可欠となりつつある Ajax 技術において，サーバからの擬似的なプッシュを実現する Comet という手法が知られている．Comet にはブラウザ間通信などさまざまな応用が考えられるが，通信の管理が複雑になるなどの問題があり，あまり一般には使われていない．そこで我々は，より簡単にブラウザ間通信を実現する手法としてタプル空間を用いた手法を提案する．タプル空間は並列プログラミングにおいてプロセス間の通信や同期のために用いられるモデルであり，それらに必要な簡潔で強力な API が定義されている．Comet は Ajax における非同期通信において，HTTP サーバからの応答をイベントの発生時まで遅延させるという手法であるが，タプル空間を使うことによってそのような通信管理をプログラマが直接扱う必要がなくなり，ブラウザ間通信をともなうアプリケーションをより簡単に作成できるようになった．本研究ではこれを，我々の開発している JavaScript 上で動作する Scheme 処理系，BiwaScheme に実装した．また，チャットシステムなどのアプリケーションの作成を通して，その効果を確かめた．

A Scheme Implementation with Inter-browser Communication Using Tuple Space

Yutaka Hara,[†1,∗1] Tomoharu Ugawa,[†1,∗2]
Taiichi Yuasa[†1] and Masahiro Yasugi[†1]

In addition to Ajax, which is an almost indispensable technique for developing modern highly-functional Web applications in JavaScript, Comet is known as an important technique to enable a server to push data to a client program. Comet has various applications such as inter-browser communication. However, it is not widely used because it needs complicated communication management. In order to implement inter-browser communication more easily by avoiding such complicated management, we propose the use of a tuple space. Tuple spaces is a model for parallel programming which is defined for communication or synchronization among processes with simple but powerful API's. Comet is a technique to delay the response of an HTTP server until an event occurs in asynchronous communication with Ajax. With a tuple space, programmers do not need to handle such communication management directly and can develop applications with inter-browser communication more easily. We implemented this model into BiwaScheme, our Scheme implementation being developed in JavaScript. We also verified the effectiveness of this system by building some applications, such as a chat system.

1. はじめに

JavaScript と非同期通信を用いて Web ブラウザ上で高機能な Web アプリケーションを実現する，いわゆる Ajax[1)] と呼ばれる技術が一般的になっている．従来の Web アプリケーションでは，新しい情報を得るには画面全体を更新する必要があったが，Ajax を用いることで画面の一部のみを更新することができるようになった．これによって，サーバに対してポーリングを行い定期的に情報を更新するようなリアルタイム性の高いアプリケーションにおいてユーザに意識させることなく通信を行えるようになり，より利便性の高いアプリケーションを作成できるようになった．

しかしポーリングには，通信の間隔を短くするとサーバに負荷がかかり，間隔を長くするとリアルタイム性が落ちるという問題がある．近年ではこの問題に対し，Ajax 技術を応用し，HTTP サーバからの応答をイベントの発生時まで遅延させることによってサーバからの擬似的なプッシュを実現する Comet という手法が知られている．Comet を利用することによって，従来の Web アプリケーションよりもはるかにリアルタイム性の高いプログラムを，ポーリングを行うことなく実装することができる．たとえばブラウザ間通信を用いて共同作業を行うアプリケーションなど，Comet にはさまざまな応用が考えられる．

しかし従来の Comet 用ライブラリは「サーバから好きなタイミングでクライアントの関数を起動する」というモデルになっているため，イベントドリブンな処理に向いている反

†1 京都大学大学院情報学研究科
Graduate School of Informatics, Kyoto University
∗1 現在，株式会社ネットワーク応用通信研究所
Presently with Network Applied Communication Laboratory Ltd.
∗2 現在，電気通信大学電気通信学部
Presently with Faculty of Electronics and Communication, The University of Electro-Communications

© 2008 Information Processing Society of Japan

面，手続き的な処理が書きにくいという問題がある．加えて，単純なデータ送信しかできないという問題もあり，Comet はあまり一般には使われていない．

そこで本研究では，より簡単にブラウザ間通信を実現する手法としてタプル空間[2)]を用いた手法を提案する．タプル空間は並列プログラミングにおいてプロセス間の通信や同期のために用いられるモデルであり，それらに必要な簡潔で強力な API が定義されている．タプル空間を使うことによって指定したデータのみ受信するなど高度な通信を行えるようになるため，ブラウザ間通信をともなうアプリケーションをより簡単に作成できる．

本研究では JavaScript 上で動作する Web アプリケーション向けの Scheme 処理系 BiwaScheme を実装し，その上にタプル空間を用いた通信システムを組み込んだ．実装にあたっては高い性能を実現するため，Scheme プログラムを中間言語にコンパイルする方式を採用した．またユーザが Web ブラウザを突然終了させる可能性を考慮して，サーバ側でタプルを管理するプロセスを動かす仕組みを用意することで，タプル空間を用いて安定したブラウザ間通信を行うことを可能にした．これによって，Scheme 言語を利用してブラウザ間通信を行う Web アプリケーションを簡単に作成できるようになった．また，チャットシステムなどのアプリケーションの作成を通して，その効果を確かめた．

本論文の構成は以下のとおりである．まず，2 章で本研究を行った背景について述べる．次に，3 章で JavaScript を用いた Scheme 処理系について説明し，4 章でタプル空間について説明する．5 章では本研究の Scheme 処理系とタプル空間の実装方法について述べる．6 章では本研究を利用した応用プログラムについて解説する．そして 7 章で今後の課題，8 章で関連研究について述べ，最後に 9 章でまとめる．

2. 研究背景

本章ではブラウザ間通信を実現するさまざまな技術について考察する．

2.1 ポーリング方式

Web ブラウザ間で通信を行うアプリケーションとしては，Web チャットが代表的なものとしてあげられる．これらのチャットシステムの多くは，一定時間ごとにユーザが「更新」ボタンを押したり，HTML の `meta` タグを利用したりするなどしてサーバに対してポーリングを行うことで通信を行っている．

この方式の利点としては実装が単純であることがあげられる．欠点としては，構造上発言がリアルタイムに更新できず，発言した内容が別のクライアントに届くまでに最悪でポーリング間隔と同じだけの遅延が発生するという問題がある．また更新があってもなくてもポーリングは行われるため，通信量が必要以上に大きくなる点もあげられる．ポーリング間隔を長くすることで通信量を減らすことは可能だが，その場合遅延がより大きくなってしまい，通信量と遅延のトレードオフになる．

2.2 プラグイン方式

Java アプレット[3)] や Adobe Flash[4)] を用いて通信を行う方式である．単純なチャットよりも，文字ではなく絵によってチャットを行う，いわゆる「お絵かきチャット」では Java アプレットや Flash によるものが多く利用されている．

この方式の利点としては，HTTP だけでなく TCP を直接使った通信が可能なため特殊なテクニックを使わなくてもサーバからのプッシュが利用可能である点があげられる．

この方式の欠点としては，実行にプラグインが必要であるため，アプリケーションが動作する環境が制限されてしまうという点があげられる．また，ユーザインタフェースに HTML を使わないためより複雑で高機能なアプリケーションを実現できる半面，通常の Web アプリケーションと操作感がまったく異なるものになりやすい，テキストデータを扱っても検索エンジンにクロールされない，アプリケーションの状態に対して URL を付けにくい，JavaScript と HTML だけで利用できる Ajax に比べると開発が手軽でないなどの問題もある．

2.3 Comet 方式

従来の HTTP 通信ではクライアントがリクエストを送ると HTTP サーバは即時にレスポンスを返すが，これを何らかのイベントが発生するときまで遅延させるなどして，HTTP では不可能なサーバからのプッシュを擬似的に実現する手法が Comet という名前で知られている．Comet を使うことによって，従来 JavaScript のみでは不可能と思われていたリアルタイムな通信が可能になるため，高い注目を集めている．この方式の利点としては，低い通信量と小さい遅延を両立できることや，一般に普及している Web ブラウザのみで実行可能な点があげられる．

Comet の問題点として，既存の Comet 用ライブラリがイベントドリブンなモデルしか提供していないため，アプリケーションによっては通信の管理が複雑になるという問題がある．

Comet では従来の HTTP 通信と違い，接続してデータを待っている状態のコネクションが多数，同時に存在することになる．そこで既存の Comet 用ライブラリは，サーバの負荷をなるべく減らすためデータ待ちのコネクションはつねに 1 本だけ張り，サーバがデータを送ってきた際にクライアントのイベントハンドラを起動する．いい方を変えれば，「サーバから好きなときにクライアントの関数を呼べる」というのが既存の Comet 用ライブラリの

© 2008 Information Processing Society of Japan

```
var turn = false;
var my_hand = null;

$("submit_button").onclick(function(){
  if(turn){
    turn = false;
    Ajax.request("/send_hand?cid=" + client_id +
                 "&hand=" + my_hand);
  }
});

function onReceive(hand, is_my_turn){
  show_hand(hand);
  if(is_my_turn)
    turn = true;
}

function onGameSet(result){
  show_result();
}
```

図 1　Comet 用ライブラリを利用したプログラムの例
Fig. 1　An example of a program with the Comet library.

提供するモデルといえる．

このモデルを利用して，トランプゲームの大富豪など各プレイヤが順番に自分の手を選ぶようなゲームの実装を考えてみる．概念的な JavaScript のプログラムを**図 1** に示す．

まず送信ボタンが押された際のイベントハンドラを登録し，次に他のプレイヤが自分の手を選んだ際にサーバから呼ばれるイベントハンドラ `onReceive`，ゲームが終了した際にサーバから呼ばれるイベントハンドラ `onGameSet` を定義している．

送信ボタンが押されると自分の選んだ手をサーバに送信するが，自分の番でない場合は手を選ぶことが許されないため，`turn` という変数を利用して現在が自分の番であるかどうかを管理している．

同じプログラムが本システムでどのように書けるかは，4.2.4 項で後述する．

3. JavaScript 上の Scheme

本章では JavaScript のかかえる問題点と，本研究で作成した Scheme 処理系がそれらをどう解決するかについて解説する．

3.1 JavaScript の問題点

3.1.1 同期処理の問題

JavaScript には，同期処理を非同期処理で書かなければならない場面がたびたび登場する．たとえば，JavaScript では以下のように for 文を用いてループを記述する．

```
for(var i=0; i<100; i++){
  //処理本体
}
```

しかしこの処理本体が時間のかかるものであった場合，for 文が終了するまで他の JavaScript のイベントハンドラなどが働かないばかりか，ブラウザによっては操作をまったく受け付けなくなるものさえある．これを避けるためには，まずループを再帰に変形し，再帰を行うところで以下のように `setTimeout` 関数を用いて処理を分断してやらなければならない．

```
var i = 0;
function loop(){
 if(i < 100){
   //処理本体
   i++;
   setTimeout(function(){ loop() }, 0);
 }
}
loop();
```

`setTimeout` は指定した時間後に与えられた関数を実行するために JavaScript に用意されている関数であるが，コンテキストスイッチを起こさせる目的にもよく利用される．

ここではまず，1 回分の処理を行う `loop` という関数を定義し，カウンタ `i` が 100 より小さければもう 1 度自分自身を呼んでループを続けるようにしている．ただし普通に呼んだだけではコンテキストスイッチが起こらず for 文と同じ結果になってしまうので，`setTimeout` 関数に自分を呼び出す処理を登録することでコンテキストスイッチを可能にしている．`setTimeout` には普通，何ミリ秒後に処理を実行するかを指定するが，ここではコンテキストスイッチを起こすのが目的であり処理を遅延させる必要はないため，時間に 0 ミリ秒を指定している．

これと同様の問題が，Ajax を用いた非同期通信の際にも存在する．たとえば，`connect` という関数がサーバからデータを取得する関数であったとすれば，「サーバからデータを取得し，それを出力する」という処理は以下のように同期的に書くのが自然である．

```
var data = connect("/data.cgi");
print(data);
```


© 2008 Information Processing Society of Japan

print 関数は与えられたデータを HTML 上に出力する関数であるとする．

しかし，JavaScript では同期通信を行う方法が存在しないため，上のような書き方は実際にはできず，以下のように「通信が完了したときに呼ばれるコールバック関数を指定する」といった回りくどい書き方をするしかない．

```
connect("/data.cgi", function(data){
  print(data);
});
```

3.1.2　ブラウザ依存の問題

JavaScript の別の問題として，ブラウザごとの仕様の違いがあげられる．たとえば非同期通信を行う関数として知られている XMLHttpRequest も Microsoft 社の Internet Explorer には存在せず，代わりに ActiveX を利用して非同期通信を行う必要がある．

また差異はライブラリ関数にとどまらず，文法上も余分なカンマをエラーとするか否かなどの違いが存在する．

3.2　BiwaScheme

BiwaScheme は本研究で実装した，JavaScript 上で動作する Scheme 処理系である．処理系の設計は Three Implementation Models for Scheme[5] を参考に，中間言語を用いたコンパイラ・インタプリタ方式を採用した．

Scheme の標準仕様である R^5RS[6]，R^6RS[7] で定義されている関数のほか，HTML 要素の操作など Web アプリケーションの作成に必要な API を定義することで，Scheme 言語を用いて Web アプリケーションを簡単に開発できるようになっている．

BiwaScheme では，非同期処理の面倒さをインタプリタで吸収することで，同期的な記述を素直に記述することが可能になっている．たとえば，上であげた setTimeout 関数を用いてループを分断する処理は，以下のように sleep 関数を用いて記述することができる．

```
(define (loop i)
  (if (< i 100)
    (begin
      ;; 処理本体
      (sleep 0)
      (loop (+ i 1)))))
(loop)
```

また「データをサーバから取得し，それを表示する」という処理は，http-request 関数を用いて以下のように記述することができる．

```
(display (http-request "/data.cgi"))
```

このほか，「ボタンがクリックされるまで待つ」などのイベントハンドラを扱う処理についても，wait-for という関数を用いて同期的に記述できる仕組みを用意している．wait-for 関数に HTML 上の要素とイベントの種類を渡すと，イベントが起こるまでインタプリタの実行が中断される．

```
(begin
  (print 1)
  (wait-for ($ "button1") 'click)
  (print 2))
```

このプログラムを実行すると，まず 1 が表示され，ボタンが押されると 2 が表示される．

もちろん，JavaScript で通常行うように非同期に起動されるイベントハンドラを定義することもできる．add-handler! という関数に HTML 上の要素とイベントの種類，クロージャを渡すと，イベントが起こった際にクロージャが実行される．

ブラウザごとの仕様の違いについては，JavaScript に用意されているライブラリ関数を Scheme 上の API として定義し直すことで差異を吸収している．同時に，プログラマが JavaScript を直接記述する必要がなくなるため，文法の差異の問題も解決されている．

4.　タプル空間

4.1　モ デ ル

タプル空間は，分散プロセス間の協調のために定義されたモデルである．このような目的のためには共有メモリなどが存在するが，タプル空間はそれらの代わりに，もしくはそれらの上に実装されたより高度なモデルとして利用される．

タプル空間は，タプルと呼ばれる値の組を表すオブジェクトの置き場である．各プロセスはタプル空間にタプルを置くことができる．また「3 つの値を持つタプル」や「値として数値の 1 を持つタプル」などの条件を指定してタプルを取り出すこともできる．

タプル空間にアクセスするための API として，Linda[2] という以下のような API が知られている．

out タプルを置く．

in 条件にマッチするタプルを探し，タプル空間から取り除く．マッチするものがなければ，マッチするタプルが置かれるまで待つ．

© 2008 Information Processing Society of Japan

inp 条件にマッチするタプルを探し，タプル空間から取り除く．マッチするものがなければ，「タプルがない」という結果を返す．

rd 条件にマッチするタプルを探し，そのコピーを返す．マッチするものがなければ，マッチするタプルが置かれるまで待つ．

rdp 条件にマッチするタプルを探し，そのコピーを返す．マッチするものがなければ，「タプルがない」という結果を返す．

Linda にはこのほかに eval という命令が定義されているが，本研究では実装していないため省略する．

タプルの具体的なデータ構造や検索条件の指定の仕方は，実装により異なる．現在ではタプル空間を扱うためのライブラリがさまざまなプログラミング言語上に実装されており，たとえば Java で実装された JavaSpaces [8] や，Ruby で実装された Rinda [9] などが存在する．

4.2　BiwaScheme のタプル空間

本研究では，ブラウザ間の通信のために Scheme 言語を用いてタプル空間を実装した．

タプル空間は Scheme 処理系 Gauche [10] のライブラリとして提供されるが，別の計算機から HTTP 経由でタプル空間にアクセスできる機構も用意されている．API は，Linda, JavaSpaces，Rinda を参考に以下のようなものを定義した．

write タプルを置く．

take 条件にマッチするタプルを探し，タプル空間から取り除いて返す．マッチするものがなければ，マッチするタプルが置かれるまで待つ．

takep 条件にマッチするタプルを探し，タプル空間から取り除いて返す．マッチするものがなければ #f を返す．

read 条件にマッチするタプルを探し，そのコピーを返す．マッチするものがなければ，マッチするタプルが置かれるまで待つ．

readp 条件にマッチするタプルを探し，そのコピーを返す．マッチするものがなければ #f を返す．

実際の関数名は，Scheme のライブラリでは tuplespace-write など tuplespace- という接頭辞を付けたものを提供している．また BiwaScheme では ts-write など ts- という接頭辞を付けたものを提供している．

4.2.1　タ　プ　ル

各タプルには，Scheme プログラムの扱える任意のオブジェクトが使用可能である．ただし，各タプルはブラウザ間の通信の際に文字列にシリアライズして送受信されるため，文字列にシリアライズできないオブジェクトや，シリアライズされた文字列から復元可能でないオブジェクトを値として使った場合の動作は保障されない．

具体的には，数値，文字列，シンボル，リスト，ベクタなどがタプルとして使用可能である．典型的な使用方法としては，数値や文字列などの値を含んだ Scheme のリストをタプルとして利用することを想定している．たとえば，100 という数値，"foo" という文字列，'abc というシンボルを含むタプルは，Scheme プログラム上で以下のように表される．

```
'(100 "foo" abc)
```

4.2.2　タプルの検索

タプルの検索には，Scheme 言語による強力な検索機能が使用可能である．

タプルを検索するための条件も，各タプルと同様に Scheme プログラム上の任意のオブジェクトが使われる．タプルが条件にマッチしたかどうかは，Gauche の提供する match マクロ [11] によって判定される．

match マクロは S 式で表される独自のクエリ言語で S 式どうしのパターンマッチを行うマクロである．たとえば「長さが 3 で，最初の値が 1 であるリスト」にマッチするパターンを以下のように書ける．

```
(1 _ _)
```

「_」はワイルドカードとして扱われ，任意の値にマッチする．

また，「?」を使うと Scheme の関数を用いて条件を指定することができる．たとえば「長さが 2 で，最初の値が数値であるリスト」にマッチするパターンは以下のように書ける．

```
((? number?) _)
```

number? はあるオブジェクトが数値かどうかを判定する Scheme の標準関数である．「?」に与えるものは評価結果が関数になるような式ならよいので，ラムダ式や高階関数を用いてより複雑な条件を指定することができる．たとえば，「1 より大きい数値」にマッチするようなパターンが以下のように書ける．

```
(? (lambda (x) (< 1 x)))
```

4.2.3　タプル管理者

Web ブラウザはユーザによって突然終了させられるなど，安定した通信を行えない可能性があるため，BiwaScheme ではサーバ側でタプル空間にアクセスする特別なクライアント（以下，タプル管理者）を動作させる仕組みを用意した．

例として，各クライアントに一意な ID を与える処理を考える．これは，各タプルの所有者を明示したいときなどに役立つ．

© 2008 Information Processing Society of Japan

各クライアントが突然終了することを考慮しなければ，この処理は以下のようなアルゴリズムによって実現できる．

(1) タプル空間サーバの起動時に，(client 0) のようなタプルをあらかじめ置いておく．
(2) 各クライアントは take を利用して，パターン (client _) にマッチするタプルを待つ．
(3) クライアントは取得したタプルの数値部分を 1 増やしてタプル空間に戻す．たとえば take が (client 3) というタプルを返したなら，数値 3 を自分の ID として覚えておき，(client 4) というタプルを write によってタプル空間に書き出す．

しかし実際には，クライアントが take によってタプルを取り除いた直後にユーザが Web ブラウザを終了させたり，ネットワークの障害が起こったりするなどの理由で，いつまでも write が実行されないという可能性が存在する．この場合，他のクライアントは take でタプルを待ち続けたままになり，プログラムの実行が続行できなくなる．

そこで本研究では，サーバ側に信頼できるプロセスとして，タプル空間にアクセスできるプログラムを置ける仕組みを用意した．このプログラムは主にタプルの管理に使われることを想定している．タプル管理者がいれば，上のような処理は以下のようなアルゴリズムで実現できる．

(1) まず，各クライアントは (client) のような 1 要素のリストをタプル空間に置く．
(2) タプル管理者は take を利用してパターン (client) にマッチするタプルを取り除き，(client 3) のようなタプルを代わりに置く．数値 3 がクライアントごとに一意な ID となる．この数値はタプルを置くごとに 1 ずつ増やすものとする．
(3) 各クライアントは take を利用してパターン (client _) にマッチするタプルを取り除き，自分の ID を得る．

1 つのタプルは 2 度以上 take または takep されることがないため，各クライアントの ID は互いに重複しない．また，クライアントがタプル (client) を置いたまま終了したり，タプル (client 3) をとり除かないまま終了しても，他のクライアントの動作に影響を及ぼすことがない．

ただしこの方法では，サーバを長時間稼動させた場合に不要なタプルが大量に残ってしまいサーバの負荷を増大させる可能性がある．この問題も，タプル管理者が不要になったタプルを一定時間ごとに取り除くようにすることで解決可能である．

タプル管理者の記述例として，上で述べたクライアントに ID を振るための管理者のプログラムを**図 2** に示す．

```
(define *clients* 0)
(add-manager
  (lambda ()
    (while #t
      (tuplespace-take *ts* '(client))
      (inc! *clients*)
      (tuplespace-write *ts* (list 'client *clients*)))))
```

図 2　タプル管理者の例
Fig. 2　An example program of a tuple manager.

まず，現在の最も大きなクライアント ID を覚えておくための変数 *clients* を 0 に初期化している．次に，本システムが提供する add-manager という関数を使って管理者プロセスを登録している．この関数は引数に Scheme の関数をとる．

この管理者の仕事は，tuplespace-take を用いてパターン (client) にマッチするタプルを取り除き，代わりに (client 3) のようなタプルを書き出すことである．ここでは Gauche の提供する while マクロを用いて無限ループを行い，そのような処理を繰り返している．inc! は Gauche の提供するマクロで，引数で指定した変数に値を 1 増やしたものを代入する．管理者プロセスからは変数 *ts* によってタプル空間にアクセスする仕様になっている．

4.2.4　タプル空間を用いたブラウザ間通信

本システムを利用して，ある Web ブラウザから別の Web ブラウザへデータを送る処理がどのように実装できるかを，簡単な例をあげて以下に示す．

まず，データを受け取る側の Web ブラウザで，以下のような Scheme プログラムを動かす．

```
(define (loop)
  (print (ts-take '(_ _ _)))
  (loop))


(loop)
```

loop という関数を定義し，自分自身を最後に呼び出すことで無限ループを行っている．無限ループの中では ts-take 関数によってタプルを取り出し，print 関数によってそれを表示している．print 関数は BiwaScheme に用意されている，HTML を用いてデータをブラウザに表示するための関数である．タプルの検索条件には，4.2.2 項で解説した長さが 3 のリストにマッチするパターンを指定している．

© 2008 Information Processing Society of Japan

`ts-take`関数は指定した条件に合うタプルが置かれるまでSchemeインタプリタの実行をブロックする．タプル空間サーバを起動した直後はタプル空間には1つもタプルが入っていないので，上のプログラムを実行しただけでは画面にはまだ何も表示されず，サーバからのレスポンスを待っている状態になる．

ここで，データを送る側のWebブラウザで以下のようなSchemeプログラムを動かす．

```
(ts-write '(1 2 3))
```

`ts-write`関数を用いて，`(1 2 3)`という長さ3のリストをタプルとしてタプル空間に書き出している．

このプログラムを実行すると，HTTPによる通信を介してタプルがサーバ側のタプル空間に置かれる．このタプルは`ts-take`関数が指定した条件にマッチするので，サーバはこのタプルをタプル空間から取り除き，受信側のWebブラウザに検索結果として返す．そして，`ts-take`関数がリスト`(1 2 3)`を返り値として返し，受信側のWebブラウザに表示される．

受信側は無限ループになっているため，すぐさま`ts-take`関数が実行され，次のタプルを待ち受ける状態になる．結果として，`ts-write`関数を実行するたびに受信側のWebブラウザに送信したタプルの内容が表示される．

タプル空間サーバはタプルが置かれるまで`ts-take`関数によるリクエストへの返答を遅延させるため，送信側がタプルを置けばすぐに受信側に結果が表示される．

また，2.3節で述べたトランプゲームの実装の例は，BiwaSchemeでは**図3**のように書ける．既存のComet用ライブラリでは自分の番かどうかを`turn`という変数を用いて管理していたが，BiwaSchemeでは相手の番の間は`ts-read`によってプログラムの実行が停止しているため，明示的に変数を用いて状態を管理する必要がない．

本システムではこのように，Cometを利用したリアルタイムな処理を非常に簡潔に書くことができる．

```
(define my-hand #f)
(let loop ()
  (wait-for ($ "submit_button") 'click)
  (ts-write (list client-id my-hand))
  (let ((hand (ts-read `(,other-client-id _))))
    (show-hand hand)
    (if (game-set?)
      (show-result hand)
      (loop))))
```

図3　本システムを利用したプログラムの例
Fig. 3　An example of a BiwaScheme program.

5. 実　　装

5.1 実装の方針

5.1.1 Scheme処理系

BiwaSchemeでは以下の点に注意して実装を進めた．

- 実用的な処理速度を実現すること．たとえば，同じくJavaScriptで書かれたScheme処理系であるjsScheme[12]では，`and`や`let`といった基本的な構文もSchemeのマクロで実装されており，長いプログラムでは実行の開始までに時間がかかるという問題があった．BiwaSchemeではできるだけ多くの部分をJavaScriptレベルで記述するなど，高い速度性能を実現するための工夫を行っている．
- 実用的なライブラリを持つこと．R^5RSやR^6RSで定義されている関数だけでなく，HTMLの操作やサーバとの通信などブラウザ上で動くという特徴を生かしたライブラリ関数を用意している．たとえば`load`という関数を使うと，サーバ上に置かれたSchemeプログラムをネットワーク経由でライブラリとして読み込むことができる．これによって，Schemeプログラムを簡単に複数のファイルに分割して記述し，より大きなアプリケーションを開発することができる．

5.1.2 タプル空間

次に，BiwaSchemeで実装したタプル空間の概要を解説する．

まず，サーバ側のプログラムは，タプル空間機能を提供するためのSchemeライブラリ，クライアントからの接続を受け付けるための簡易的なHTTPサーバ，クライアントからのリクエストをタプル空間へのリクエストに変換するプログラムから構成される．クライアント側のプログラムは，サーバ側のタプル空間にアクセスするためのJavaScriptで書かれた処理と，BiwaSchemeからそれらの処理を呼び出すための定義からなる．

サーバ側はApache[13]など既存のHTTPサーバを利用するという案もあったが，以下のような理由から独自に実装することを決定した．まず，HTTPを用いた通信を行うプログラムを書く最も簡単な方法はCGIスクリプトとして実装することであるが，単純なCGIスクリプトだけではタプル空間を実装することができない．なぜならば，CGIスクリプトはクライアントからリクエストがあるたびに起動されるが，タプル空間に置かれたデータを

© 2008 Information Processing Society of Japan

記憶しておくには，クライアントからのリクエストがない間も動作し続けているプロセスが必要だからである．もちろん，データベースなどを用いてリクエストごとにタプル空間内のデータをシリアライズすることでもこの問題を解決することができるが，性能面で問題がある．

よって，考えられる選択肢は以下のいずれかとなる．

- HTTP サーバは既存のものを利用し，サーバに常駐するプロセスと，それにアクセスするための CGI スクリプトを実装する．
- HTTP サーバを独自に実装し，そこにタプル空間を組み込む．

実装が手軽なのは前者であるが，本研究ではスケーラビリティ対応など，将来の拡張を考えて後者を選択した．

5.2 Scheme 処理系

本節では BiwaScheme の実装について解説する．

5.2.1 データ構造

BiwaScheme では，Scheme プログラムで使われる各種のデータを JavaScript のクラスを用いて表す．JavaScript はプロトタイプベースのオブジェクト指向言語であり，「クラス」や「メソッド」といった用語を用いるのは正確とはいえないが，プロトタイプ機構をうまく使うことにより Java や C++などのようなクラスやメソッドを擬似的に実現できることが知られており，以下でも簡単のためこれらの用語を使用することにする．

BiwaScheme で定義しているクラスには以下のようなものがある．

`BiwaScheme.Symbol` Scheme のシンボルを表す．

`BiwaScheme.Pair` Scheme のコンスセルを表す．

`BiwaScheme.Char` Scheme の文字を表す．

`BiwaScheme.Port` Scheme のポートを表す．

`BiwaScheme.Syntax` Scheme の構文，マクロの変換器を表す．

各クラスは，他の JavaScript ライブラリとの衝突を避けるため，トップレベルにではなく `BiwaScheme` というオブジェクトのプロパティとして宣言されている．空のリストを表すオブジェクトも同じ名前空間に定義されており，`BiwaScheme.nil` という名前が付いている．`nil` は `Pair` のインスタンスになっている．

Scheme プログラムに登場するオブジェクトのうち，文字列や数値などは JavaScript のオブジェクトをそのまま表現に使用している．これによって，余計なオブジェクトのインスタンスが生成されるのを防ぎ，より性能を上げることができる．

Scheme の値と JavaScript の値の対応を以下に示す．

ベクタ　JavaScript の配列で表現する．

数値　JavaScript の数値で表現する．ただし，`+inf.0`，`-inf.0` はそれぞれ `Infinity`，`-Infinity` で，`+nan.0` は `NaN` で表す．

文字列　JavaScript の文字列で表現する．

真偽値　JavaScript の真偽値で表現する．

未定義値　JavaScript の `undefined` で表現する．

このうち配列は，ベクタの表現だけでなくコンパイルされた中間言語や Scheme のクロージャの表現としても使われる．

JavaScript にはこのほかにも `null` オブジェクトが存在するが，BiwaScheme では意図的に使用していない．たとえば，Scheme では空リストの car や cdr を参照することが許されていないため，`nil` は必ずしも `Pair` のインスタンスである必要はなく，`null` を空リストの JavaScript 表現として用いることも可能ではある．しかし，JavaScript では `==` のような演算子を使用した場合に自動的に型変換が試みられるため，たとえば以下の式が真になる．

```
null == undefined
```

このため，発見しにくい不具合を生む可能性があると判断し，`null` は Scheme オブジェクトの表現としては使わないことを決めた．

Scheme の関数は，JavaScript の関数か，クロージャオブジェクトとして表現される．ライブラリ関数のほとんどは前者で表現される．ユーザが定義した関数はすべて後者で表現される．クロージャオブジェクトは関数の中間言語表現とそれが参照する自由変数を 1 つにまとめたもので，実装上は JavaScript の配列を用いている．

5.2.2 実 行 方 式

BiwaScheme では中間言語方式を採用しており，Scheme のプログラムをまず中間言語にコンパイルしてからインタプリタで実行するようになっている．

Scheme のソースプログラムを実行する手順は以下のとおりである．

(1) S 式の構文解析を行い，`Pair` や `Symbol` といったクラスのインスタンスを用いて構文木を構成する．
(2) マクロの展開を行う．
(3) 展開結果の構文木をコンパイルし，中間言語表現を生成する．
(4) 中間言語表現をインタプリタで実行する．

© 2008 Information Processing Society of Japan

(5)　実行結果を JavaScript の値として返す．

ただし，後述する `Pause` クラスを使用してインタプリタの実行が分断される場合があるため，実行結果は単純に関数の返り値として返すことはできない．そのため，すべての式の評価が終わったタイミングで実行されるコールバック関数を指定することで実行結果の受け渡しを行うという仕様になっている．

5.2.3　コンパイラ

コンパイラは，パーザの作った構文木から中間言語表現を生成する．3.2 節で述べたように，コンパイラや中間言語，インタプリタの仕様は Three Implementation Models for Scheme[5] を参考にしている．たとえば，`set!` によって代入されうる変数の値をスタックに置かないという工夫はこの論文を参考にした．

BiwaScheme では，スタックのコピーを作ることで継続オブジェクトを作り，それをスタックに復元することで継続を実行するという設計になっている．しかしコピーを作ったあとに，コピー内に含まれる変数の値が `set!` により書き替わった場合，コピーの方も値を更新しないと継続を正しく再開することができない．しかし，コピー内のすべての変数の値の変更を監視するのは容易ではない．

そこで，コンパイル時に `set!` によって値が変更される可能性のある変数をリストアップし，これらの変数を参照するときは値をスタックに置くのではなく，値を box 化したものをスタックに置くようにする．box 化は長さが 1 の配列に値を入れるという作業である．変数の参照および代入をこの box を通して行うことで，スタックのコピーと同期をとる必要がなくなり，より安全かつ高速に継続と代入をサポートすることができる．

5.2.4　インタプリタ

BiwaScheme ではレジスタとスタックを用いてプログラムの実行を制御する．レジスタは `a`，`x`，`f`，`c`，`s` という名前の JavaScript の変数であり，スタックは JavaScript の配列である．JavaScript では，C 言語などのように配列の長さをあらかじめ決めておく必要がなく，実行時に長さを伸ばすことが可能であるため，Scheme プログラムはメモリが許す限りの深さのスタックを使用できる．

JavaScript は静的な型を持たないため，スタックには JavaScript の任意のオブジェクトを入れることができる．具体的には，関数の引数として渡される値や，関数の実行前に保存されるレジスタの値などがスタックに積まれる．

Scheme プログラムの値が格納される場所には，レジスタとスタックのほかに `TopEnv` という変数がある．`TopEnv` にはグローバルな変数束縛が格納される．

5.2.5　中 間 言 語

BiwaScheme では JavaScript の文字列，数値，配列などを用いて中間言語を表現している．たとえば，以下のような短い Scheme プログラムを考える．

```
(print 1)
```

これは以下のような中間言語表現にコンパイルされる．

```
["frame",
   ["constant", 1,
   ["argument",
   ["refer-global", "print",
   ["apply", 1]]]],
["halt"]]
```

中間言語表現は命令を表す配列がネストした構造になっている．1 つの配列が 1 つの命令を表し，配列の最初の要素が命令の種類を表す文字列になっている．配列の長さは命令の種類によって決まっている．`halt` や `apply` などを除くほとんどの命令は，配列の最後に次の命令へのポインタが格納される．インタプリタはこのポインタを次々にたどることでプログラムを実行していく．

5.2.6　標準関数の実装

BiwaScheme は現在，R^6RS への準拠を目標に開発しており，多くの標準関数と，一部の標準ライブラリが実装済みである．

標準関数のうち `map` や `for-each` など Scheme のクロージャを呼び出す関数を JavaScript で実装するために，`Call` という特別なクラスを実装した．たとえば，`for-each` はリストの各要素に対して指定したクロージャを実行する関数であるが，リストの各要素に対して繰り返す JavaScript のループと，インタプリタの実行ループを並列に走らせることはできないので，何らかの工夫が必要となる．クロージャの呼び出しごとに新しいインタプリタを呼び出すのは方法の 1 つであるが，Scheme ではクロージャの呼び出し中に継続のキャプチャが起こる可能性があるためこの方法をとることはできない．

そこで BiwaScheme では，`Call` というオブジェクトを経由してあるクロージャを実行したいことをライブラリ関数からインタプリタへと伝える．`Call` オブジェクトには実行したいクロージャ，引数，実行が終わった際に呼ばれる JavaScript のコールバック関数を含めることができる．

インタプリタは，ライブラリ関数が実行結果として `Call` クラスのインスタンスを返す

© 2008 Information Processing Society of Japan

と，「指定されたクロージャを呼び出し，その結果を引数にコールバック関数を実行する」という処理を表す中間言語表現を動的に生成し実行する．

これを利用して，map や for-each など，リストの各要素についてクロージャを呼び出すような処理が以下のように書ける．まず，1 ステップごとに計算を中断できるように，リストの各要素について繰り返す処理を CPS 形式で記述する．各ステップでは，実行したいクロージャと引数となるリストの要素，残りの計算を表す関数を Call オブジェクトに保存しインタプリタに返す．リストの末尾にたどり着いた際は Call オブジェクトではなく計算結果をただ返すことで，繰返しを終了できる．ここではリストを例にとったが，もちろん vector-map などリスト以外のものについて繰り返しクロージャを実行するようなものも同様に実装できる．

5.2.7　ブラウザ向け関数の実装

BiwaScheme では，3.2 節であげたように，Web アプリケーション用のさまざまな関数を実装している．

これらの API のうち，sleep や http-request などを実装するためには setTimeout や XMLHttpRequest のような非同期処理を行う JavaScript の関数を使用しなければならない．これらの関数は，一定時間の経過や通信の終了などのイベントが起こった際のコールバック関数を指定する仕様になっている．よって，Scheme インタプリタの実行とこれらの関数を組み合わせて使うには，Scheme プログラムの実行処理を中断・再開する機構が必要となる．BiwaScheme では Pause という特殊なクラスを用意している．

Pause はインタプリタのある瞬間の実行状態を表すオブジェクトで，レジスタの値などが保存されている．Pause クラスには resume というメソッドが定義されており，これを呼び出すとインタプリタの実行が再開され，Scheme プログラムの残りの部分が評価される．

これを用いて，たとえば sleep 関数を**図 4** のように実装することができる．

sleep 関数は BiwaScheme で定義した，指定した時間だけ Scheme インタプリタの実行を停止する関数である．JavaScript 上では，Pause オブジェクトを利用して Scheme インタプリタの実行を中断し，指定した時間後にインタプリタの実行を開始するようなコールバック関数を setTimeout 関数に渡すことで停止と再開の処理を行っている．

```
define_libfunc("sleep", 1, 1, function(ar){
  var msec = ar[0];
  return new BiwaScheme.Pause(function(pause){
    setTimeout(function(){ pause.resume() }, msec);
  });
});
```

図 4　sleep 関数の実装コード
Fig. 4　The implementation code for the sleep function.

define_libfunc は，BiwaScheme のライブラリ関数を JavaScript で定義する際に使われる関数で，関数名，引数の最小・最大の個数，処理の本体を引数にとる．処理の本体は JavaScript の関数で，呼び出された際の引数を格納した配列 ar を引数として受け取る．sleep 関数は 1 引数の関数なので，ar[0] から時刻を取得している．次の行では，新しい Pause オブジェクトを生成して返している．生成時の引数には JavaScript の関数を渡す．この関数には，インタプリタの状態を保存した Pause オブジェクトが引数として渡され，その resume メソッドを適切なタイミングで呼び出すことでインタプリタを再開できる．ここでは，setTimeout に与えるコールバック関数の中で resume メソッドを呼び出している．

Pause オブジェクトによる実行の中断・再開は sleep や http-request だけでなく，サーバとの非同期通信を行う関数全般にも利用できる．たとえばタプル空間との通信や，5.1.1 項で述べた load 関数も Pause オブジェクトを利用して実装されている．

5.3　タプル空間

本節では本研究におけるタプル空間の実装について解説する．

5.3.1　サーバ側ライブラリ

まず，Scheme 言語のライブラリとしてのタプル空間の実装について説明する．実装にはモジュールやスレッドなど，他の Scheme 処理系と互換性のない Gauche 独自の機能を利用しているため，このライブラリを利用するには Gauche が必要である．

タプル空間ライブラリは tuplespace という名前のモジュールとして実装した．このモジュールをインポートすると，以下の関数が利用できるようになる．

- tuplespace-init
- tuplespace-write
- tuplespace-read
- tuplespace-readp
- tuplespace-take
- tuplespace-takep
- tuplespace-dump
- tuplespace-clear

それぞれの機能は以下のとおりである．

- tuplespace-init は新しいタプル空間を作成して返す．タプル空間は <tuplespace>

© 2008 Information Processing Society of Japan

クラスのインスタンスで表現される．タプル空間オブジェクトは保持しているタプルのリストと，`read` や `take` によって登録されるコールバック関数を保持している．
以下のタプルスペースにアクセスする関数はすべて，このタプル空間オブジェクトを引数にとる．`tuplespace-init` を複数回呼ぶことによって，独立した複数のタプル空間を使用することも可能である．ただしタプル空間サーバおよび BiwaScheme によるクライアントは今のところ単一のタプル空間しかサポートしていない．

- `tuplespace-write` はタプル空間にタプルを書き出す．タプルとしては Scheme の任意の値が利用可能である．
- `tuplespace-read` はタプル空間からタプルを検索する．検索クエリは S 式で与える．クエリの仕様については 4.2.2 項で述べたとおりである．条件にマッチするタプルが見つかったら，そのタプルの参照を返す．見つからなければ，スレッドを用いて見つかるまで待つ．
- `tuplespace-readp` はタプル空間からタプルを検索する．条件にマッチするタプルが見つかったら，そのタプルの参照を返す．見つからなければ `#f` を返す．
- `tuplespace-take`，`tuplespace-takep` は `tuplespace-read`，`tuplespace-readp` と同じであるが，見つかったタプルをタプル空間から取り除くことが異なる．
- `tuplespace-dump` はタプル空間に存在するすべてのタプルを文字列として表現したものを返す．主にデバッグのために用意されている．
- `tuplespace-clear` はタプル空間に存在するすべてのタプルを取り除き，タプル空間を空にする．`tuplespace-takep` を `#f` が返るまで繰り返し実行することによっても同様の処理が可能であるが，簡単のために用意されている．主にユニットテストのために用意されている．

5.3.2　サ　ー　バ

タプル空間サーバもライブラリと同じく，Gauche と Scheme 言語を用いて開発した．
まず，Scheme 言語で書かれた簡易的な HTTP サーバを実装した．この HTTP サーバには以下のような特徴がある．

- 200 行程度と，非常に小さい実装になっている．その分，正常でないリクエストへの対処が不十分であるなど，HTTP サーバとしての機能は限定されている．
- URL で指定したファイルをレスポンスとして返すことができる．これは HTML ファイルを表示するために必要である．
- ある正規表現にマッチする URL へのリクエストがあったときに実行される処理をコールバック関数として指定できる．この機能は，たとえば `http://`*host*:*port*`/time` という URL にアクセスしたら現在の時刻が書かれた HTML データを返すようにするなど，一般の Web サーバにおける CGI スクリプトのように使用できる．

さらに，このコールバック関数を登録する機構を用いて，HTTP 経由でタプル空間にアクセスできる機能を実装した．たとえば，`http://`*host*:*port*`/ts/write?(1 2 3)` にアクセスするとタプル空間にタプル `(1 2 3)` を書き込むことができる．
`read` や `take` に対応する URL にアクセスすると，サーバはタプルが見つかるまで待ち，タプルが見つかったらすぐにレスポンスを返す．これは，コールバック関数の内部で `tuplespace-read` や `tuplespace-take` を実行することで実装している．これによって，Comet を用いたリアルタイムな通信を実現している．
タプル管理者を登録するために，4.2.2 項で解説した `add-manager` 関数を用意した．この関数に Scheme の関数を渡すことで，指定した処理を行うタプル管理者を動作させることができる．渡された関数は独立したスレッドで実行するので，複数のタプル管理者を動作させることができる．

5.3.3　クライアント

タプル空間クライアントは，BiwaScheme 上の API として実装した．`ts-write`，`ts-read` などの関数を用いてタプル空間へのアクセスを行うことができる．
これらのライブラリ関数は JavaScript で実装されており，内部では `Pause` オブジェクトを用いることでサーバとの通信が完了するまでインタプリタの実行を中断している．

6. 応用プログラム

6.1　チャットシステム

BiwaScheme を利用したブラウザ間通信を行うプログラムの例として，簡単なチャットシステムを作成した．チャットのプログラムを**図 5**，**図 6** に，実行画面を**図 7** に示す．
自分の名前と発言したいメッセージをテキストボックスに書き込み，送信ボタンを押すと発言内容がサーバに送られる．発言はタプル空間を介して，同じページを見ている他の参加者にリアルタイムに送信される．

6.1.1　実　　装

各クライアントからの発言を保持するためにタプル空間を利用している．
タプル空間に置かれる各タプルは 5 つの要素を持つリストで，以下のような構造をしている．

© 2008 Information Processing Society of Japan

```
(define message-id 0)
(define client-id 100)

;; クライアント接続時に，クライアント ID と最新の発言 ID を教える
(add-manager
  (lambda ()
    (while #t
      (tuplespace-take *ts* '(connect))
      (inc! client-id)
      (tuplespace-write *ts*
        `(connect ,client-id ,message-id)))))

;; 送られてきた発言にメッセージ ID を付加する
(add-manager
  (lambda ()
    (while #t
      (let ((tuple (tuplespace-take *ts* '(message _ _ _))))
        (inc! message-id)
        (tuplespace-write *ts*
          `(message ,message-id ,@(cdr tuple)))))))
```

図 5　チャットのサーバ側プログラム

Fig. 5　The server program for the chat system.

(message 発言 ID 発言者 ID 発言者名 発言内容)

各要素は以下のような意味を持つ．

message　発言を表すタプルであることを示す．

発言 ID　各発言ごとに一意な整数が割り当てられる．各クライアントは，これを利用して発言の未読管理を行う．

発言者 ID　各クライアントごとに一意な整数が割り当てられる．各クライアントは，これを利用して自分の発言かどうかを判断することができる．

発言者名　発言者の名前がここに入る．

発言内容　発言内容の文字列がここに入る．

サーバ側では，発言者 ID を割り当てるものと発言 ID を割り当てるものの 2 種類のタプル管理者プロセスを動作させている．

まず，4.2.3 項で説明したアルゴリズムを用いて各クライアントに一意な発言者 ID を割り当てている．まず，クライアントは (connect) のようなタプルをタプル空間に書き出す．サーバ側のタプル管理者はこのようなタプルをつねに検索しておき，これを見つけたら (connect 発言者 ID 発言 ID) のようなタプルに置き換える．クライアントは (connect _ _) のようなパターンを用いてこれらの情報を受け取る．ここでは発言者 ID に加え，一番新しい発言の発言 ID もクライアントに渡している．これはチャットに途中参加するクライアントのためである．

```
(define *my-id* #f)
(define *msg-id* #f)

(add-handler! ($ "send") "click"
  (lambda ()
    (let ((name (get-content ($ "name")))
          (msg  (get-content ($ "message"))))
      (ts-write (list 'message *my-id* name msg)))))

(define (receive-message)
  (let ((v (ts-read `(message (? (lambda (x) (= (+ 1 ,*msg-id*) x))) _ _ _))))
    (let ((new-msg-id (cadr v))
          (sender-id  (caddr v))
          (name       (cadddr v))
          (message    (cadddr (cdr v))))
      (set! *msg-id* new-msg-id)
      (print "<" name ">" message)))
  (receive-message))

(define (start-chat)
  (ts-write '(connect))
  (let ((state (ts-take '(connect _ _))))
    (set! *my-id* (cadr state))
    (set! *msg-id* (caddr state)))
  (receive-message))

(start-chat)
```

図 6　チャットのクライアント側プログラム

Fig. 6　The client program for the chat system.

発言 ID も発言者 ID と同じくタプル管理者を用いて付加されている．まず，クライアントは発言時に以下のようなタプルを書き出す．

(message 発言者 ID 名前 発言内容)

サーバ側のタプル管理者はこれを以下のようなタプルに置き換える．

(message 発言 ID 発言者 ID 名前 発言内容)

クライアント側では，ts-read を使ってこのようなタプルを取得する．サーバは ts-read によるリクエストに対し，タプルが見つかったときに初めてレスポンスを返すので，他人の発言に対しリアルタイムな更新が実現できる．また，最後に読み込んだタプルの発言 ID を記憶しておくことで，同じ発言を何度も読み込んでしまわないようにしている．

6.1.2　評　　価

本研究のシステムを用いることによって，サーバ側・クライアント側合わせて 50 行程度と少ない行数でリアルタイムなチャットシステムを作成することができた．

© 2008 Information Processing Society of Japan

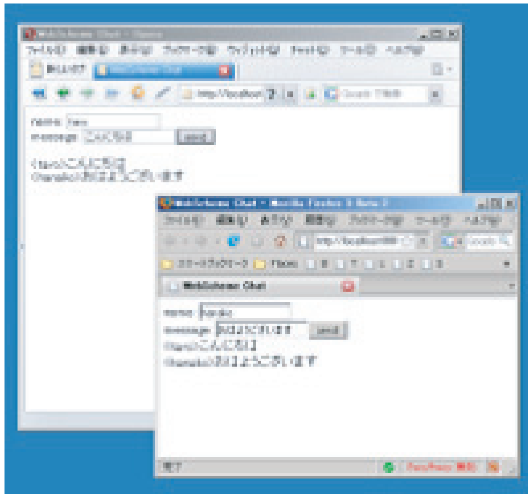

図 7　チャットプログラムの実行画面
Fig. 7　A screenshot of the chat system.

7. 今後の課題

7.1 デバッグ機能の強化

現在の BiwaScheme ではエラーが発生したときに行番号などを表示する機能がない．エラー時に適切な行番号やスタックトレースを表示できるようにすることで，よりプログラムの開発効率を高めることができる．

7.2 標準への準拠

syntax-rules を用いた衛生的なマクロや，有理数・複素数のサポートなど，R^6RS に準拠した処理系となるにはまだ不足している機能が多く残っている．これらを実装し，より完成度の高い処理系を目指すことで，他の Scheme 処理系の利用者にも安心して利用してもらえるものにしたい．

またライブラリについても，5.2.6 項で述べたとおり R^6RS のライブラリのごく一部しか実装できていない．残りのライブラリにはバイトベクタやファイルシステムへのアクセスなど JavaScript では実装が困難なものも含まれるが，ハッシュテーブルや列挙型などは JavaScript でも実装することができ，利用価値も高いと考えられる．

また Scheme には SRFI[14] と呼ばれるライブラリ規格があり，すでに 60 を超えるライブラリの仕様が策定されている．これらのうち，たとえばリストや文字列に関するライブラリを定めたものなどは特に有用であり，BiwaScheme でもサポートしたい．

7.3 ライブラリの自動ロード

BiwaScheme によるアプリケーションは通常，ネットワーク経由でダウンロードされ実行されるため，ファイルサイズが大きくなるとアプリケーションの実行まで時間がかかり，好ましくない．しかし，現在の実装では BiwaScheme のロード時にすべてのライブラリを読み込むため，プログラムで使用していないライブラリも読み込んでしまうという問題がある．

必要なライブラリを自動的に読み込む機構を実装することで，この問題を解決することができる．

7.4 クロージャの移送

中間言語表現を文字列にシリアライズし，これをサーバや他の Web ブラウザに送信することで，クロージャや継続をネットワーク上でやりとりできるようになり，さまざまな応用が考えられる．

ただし `set!` により代入が起こる場合など，単純に移送が行えない場合もあり，このようなケースをどう扱うかについて検討する必要がある．

7.5 タプル空間の機能強化

7.5.1 トランザクション

4.2.3 項ではタプル管理者の仕組みを利用してクライアントがタプルを「持ち逃げ」しても問題のないシステムを実装する例を示した．しかしタプルを別のものに置き換えるという処理はさまざまなアプリケーションで使われるため，より簡単な方法で `take` と `write` をアトミックに実行できれば便利である．

JavaSpaces では，トランザクション機能を提供することによってこの問題に対処している．たとえば `take` によるタプルの取り出しと `write` によるタプルの書き出しを 1 つのトランザクション中で行うと，`write` が正常に実行されて初めて実際のタプルの置き換えが行われる．よって，`take` を実行した直後にクライアントプログラムが終了してもタプルは保存される．

BiwaScheme でも同様の機能を提供することで，信頼性の高い通信を簡単に実装できるようになると思われる．

© 2008 Information Processing Society of Japan

7.5.2　生存期間の指定

本システムでは，タプル管理者を利用して不要になったタプルを掃除することが可能である．しかしこのような処理はよく使われるため，タプルを `write` でタプル空間内に書き出す際にタプルの生存期間を指定し，生存期間を過ぎたタプルは自動的に消去されるようにする機構があると便利だと思われる．

7.5.3　タイムアウト

現在，`read` や `take` を行うと，プログラムはタプルが見つかるまで永遠に待ち続けるが，この仕様では一定時間が過ぎたらエラーにするなどの処理を記述することが難しい．これは，JavaSpaces や Rinda が実装している，`read` や `take` での検索時にタイムアウトする時間を指定することができる機能を取り入れることで解決できる．

7.5.4　セキュリティ

現在，BiwaScheme が用意しているタプル空間サーバはセキュリティについて考慮していない．このため，悪意のあるクライアントから大量のタプルを送り付けられたり，逆に任意のタプルを `takep` し続けられたりすると，アプリケーションは正常に動作しなくなる．

タプル管理者を置くことによってこのような攻撃に対処することもできるが，クライアントのプログラムが改竄されていないかどうかをチェックするなど，より手軽に安全なアプリケーションを開発できるような仕組みが望まれる．

またタプルの検索クエリにラムダ式を記述できるため，サーバ側で危険なコードが実行されてしまう可能性がある．検索クエリの S 式中で使用できるシンボルを制限するなどの対策を検討中である．

8. 関 連 研 究

8.1　Links

Links[15] はエディンバラ大学で開発されている Web アプリケーションのためのプログラミング言語で，1 つのソースコードからサーバ用，クライアント用，データベース用のプログラムを生成するのが特徴である．

サーバ側とクライアント側のプログラミング言語を同じにすることでプログラマの負担を軽くするという思想は本研究と類似している．本研究では，Links が実現していないブラウザ間の通信を簡単に行う手段を用意している．

8.2　Thread.Concurrent

Thread.Concurrent[16] は JavaScript 上でマルチスレッドを実現するためのライブラリである．すべてが JavaScript のみで実現されているため，プラグインなど特別な環境を必要とせず，一般に普及している Web ブラウザのみで動作することが特徴となっている．

Thread.Concurrent には JavaScript で書かれた JavaScript の字句解析器・構文解析器が含まれており，これを用いて与えられたソースプログラムを `setTimeout` を用いてスケジューリングするプログラムにコード変換することによりマルチスレッド化を実現している．

2 章において JavaScript の問題の 1 つに同期処理を非同期処理に手動で変換しなければならないことをあげたが，Thread.Concurrent はこれに対する解決策となっている．処理系の動作に JavaScript 以外を必要としないのは BiwaScheme と同じであるが，まったく違うアプローチとなっている点が興味深い．

9. お わ り に

本研究では，従来プログラマが複雑な通信の管理を強いられてきた Comet によるリアルタイムなブラウザ間通信において，タプル空間を用いることで簡潔な記述でブラウザ間の通信を行う手法を提案した．また，Scheme 言語用のタプル空間ライブラリや簡易 HTTP サーバを実装し，このシステムを実際に我々の開発している Scheme 処理系，BiwaScheme 上に実装した．

さらに，実際にリアルタイムなブラウザ間通信を行うアプリケーションの例としてチャットシステムを実装し，BiwaScheme を用いてこのようなアプリケーションが簡単に作成できることを示した．

今後は処理系の性能の改善や機能の充実など，実際のアプリケーションの作成に役立つ改善を行っていきたい．

謝辞　本研究の一部は，情報爆発に対応する高度にスケーラブルなソフトウェア構成基盤（18049015）（科学研究費特定領域研究「情報爆発時代に向けた新しい IT 基盤技術の研究」）の補助を得て行った．

参 考 文 献

1) Garrett, J.J.: A New Approach to Web Applications. http://www.adaptivepath.com/publications/essays/archives/000385.php
2) Gelernter, D.: Generative communication in Linda, *ACM Trans. Programming Languages and Systems* (*TOPLAS*), Vol.7, No.1, pp.80–112 (1985).
3) ColinFraizer, J.: JAVA API リファレンス — java.applet および java.awt，プレンティスホール出版 (1996).

© 2008 Information Processing Society of Japan

4) 大津　真：ActionScript 3.0 プログラミング入門 — for Adobe Flash CS3，ビー・エヌ・エヌ新社 (2008).
5) Dybvig, R.K.: Three Implementation Models for Scheme, Ph.D. Thesis, University of North Carolina (1987).
6) Kelsey, R., Clinger, W. and Rees, J.: Revised^5 Report on the Algorithmic Language Scheme, *Higher-Order and Symbolic Computation*, Vol.11, No.1 (Aug. 1998).
7) Sperber, M., Dybvig, R.K., Flatt, M. and Straaten, A.V.: Revised^6 Report on the Algorithmic Language Scheme. http://www.r6rs.org/final/html/r6rs/r6rs.html
8) Sun Microsystems: JavaSpace Specification (1998).
9) 関　将俊：dRubyによる分散・Webプログラミング，アスキー (2005).
10) 川合史朗，Kahuaプロジェクト：プログラミングGauche，オーム社 (2008).
11) 川合史朗：Gaucheリファレンスマニュアル 11.46 util.match — パターンマッチング.
12) Yakovlev, A.: jsScheme – Scheme interpreter in JavaScript (2003). http://alex.ability.ru/scheme.html
13) The Apache Software Foundation: The Apache HTTP Server Project. http://httpd.apache.org/
14) The SRFI Editors: Scheme Requests for Implementation. http://srfi.schemers.org/
15) Cooper, E., Lindley, S., Wadler, P. and Yallop, J.: Links: Web ProgrammingWithout Tiers, *FMCO '06* (2006).
16) 牧　大介，岩崎英哉：非同期処理のためのJavaScriptマルチスレッドフレームワーク，情報処理学会論文誌：プログラミング，Vol.48, No.SIG 12(PRO 34), pp.1–18 (2007).

(平成20年2月18日受付)
(平成20年4月 8 日採録)

原　　悠（正会員）

1983年生．2006年京都大学工学部情報学科卒業．2008年京都大学大学院情報学研究科修士課程修了．現在，（株）ネットワーク応用通信研究所に勤務．プログラミング言語に興味を持つ．

鵜川　始陽（正会員）

1978年生．2000年京都大学工学部情報学科卒業．2002年京都大学大学院情報学研究科修士課程修了．2005年京都大学大学院情報学研究科博士後期課程修了．同年京都大学大学院情報学研究科特任助手．2007年同研究科特任助教．2008年電気通信大学助教．博士（情報学）．プログラミング言語処理系に興味を持つ．日本ソフトウェア科学会会員．

湯淺　太一（フェロー）

1977年京都大学理学部卒業．1982年京都大学大学院理学研究科博士課程修了．同年京都大学数理解析研究所助手．1987年豊橋技術科学大学講師．1988年同大学助教授，1995年同大学教授，1996年京都大学大学院工学研究科情報工学専攻教授．1998年同大学院情報学研究科通信情報システム専攻教授．理学博士．記号処理，プログラミング言語処理系に興味を持っている．著書『Common Lisp 入門』（共著），『C言語によるプログラミング入門』，『コンパイラ』ほか．ソフトウェア科学会，電子情報通信学会，IEEE，ACM各会員．

八杉　昌宏（正会員）

1967年生．1989年東京大学工学部電子工学科卒業．1991年東京大学大学院電気工学専攻修士課程修了．1994年同大学院理学系研究科情報科学専攻博士課程修了．1993〜1995年日本学術振興会特別研究員（東京大学，マンチェスター大学）．1995年神戸大学工学部助手．1998年京都大学大学院情報学研究科通信情報システム専攻講師．2003年同大学助教授．2007年より同大学准教授．博士（理学）．1998〜2001年科学技術振興事業団さきがけ研究21研究員．並列処理，言語処理系等に興味を持つ．日本ソフトウェア科学会，ACM各会員．

© 2008 Information Processing Society of Japan