移動計算機環境を支援する言語 LEAD とそのメタレベルアーキテクチャ

天野憲樹, 渡部卓雄

1996 年 5 月 31 日
IS-RR-96-0014S

北陸先端科学技術大学院大学
情報科学研究科
〒 923-12 石川県能美郡辰口町旭台 15
{n-amano,takuo}@jaist.ac.jp

©Noriki Amano, Takuo Watanabe, 1996

ISSN 0918-7553

要 旨

移動計算機環境では計算機の移動等にともない、システム構成や環境が動的に変化する。従来のシステム構成技術 (OS、言語等) では、このような計算機環境に柔軟に対応可能なアプリケーシ ョンを作成することは困難である。これまでシステム構成や環境の動的な変化に対し、OS レベルもしくはそれ以下のレイヤで、変化に対応することが一般的であ った。しかしアプリケーシ ョンより下位のレイヤで論理的なシステム透過性を実現しても、移動計算機環境に内在する物理的な制約から解放されるわけではない。このような問題に対応する枠組の一つとして、アプリケーシ ョンそのものがシステム構成や環境に応じて振舞いを変える、ということが考えられる。これは状況に適した処理を行なうという観点から望ましいばかりでなく、内在する制約の緩和に貢献することも期待できる。本稿ではこうした移動計算機環境に柔軟に対応可能なアプリケーシ ョンを作成 ・支援する枠組として、メタレベル ・アーキテクチャを用いた言語処理系を提案する。

1 はじめに

1.1 背景

近年の技術革新は電子装置の小型化、高性能化をもたらし、その製造技術の進歩はそれらの安価な実現を可能にした。その結果、パーソナルユースの携帯電話、ポケットベル等が急速に普及するこことなった。またネットワークは拡大の一途を辿り、オフィスはもとより、パソコン通信などを通じて急速に個人の生活空間に浸透しつつある。そして無線通信の技術革新により携帯可能な小型計算機の実用性が高まってきた。このような携帯可能な小型計算機とネットワーク、そして無線通信という 3 つの技術の融合により新しいコンピューティング環境、コンピューティングスタイルが生まれようとしている。それは今までのコンピュータの利用形態とは全く異なる利用が可能になり、今までにないコンピューティングスタイルが生み出される可能性を含んでいる。このように移動可能な小型計算機をネットワークに接続し、分散環境と融合させた計算機環境を移動計算機環境 [2] という。

このような移動計算機環境ではその時々の計算機の使用形態、使用場所、使用目的に応じて、計算機システムの構成や環境が動的に変化する。しかしその上で動作するソフトウエアは静的であり、固定化されている。これが移動計算機環境におけるシステムの透過性やアプリケーションの効率的実行を困難にしている要因であると言えよう。しかし現在このような環境に対応可能なソフトウエアが望まれている。動的なシステム構成の変化には、やはりそれに応じて動的にその振舞いを変えるソフトウエアが望ましい。状況に応じて自らの振舞いを変えるソフトウエアの実現にはリフレクションやメタレベル・アーキテクチャ [6] といった技術の応用が可能である。

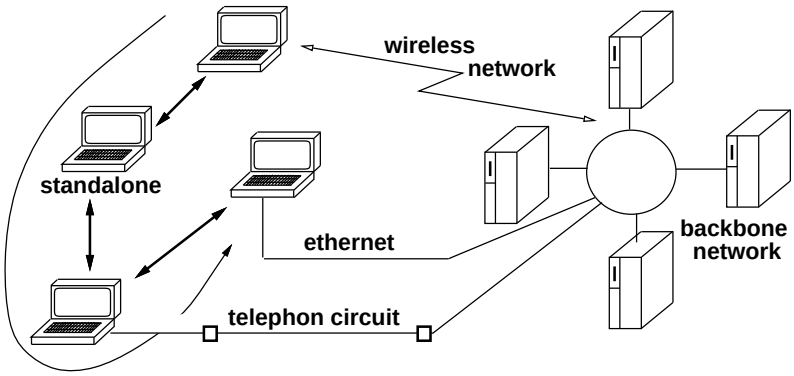

図 1: システム構成の動的な変化

1.2 本研究のアプローチ

移動計算機環境に対する従来の研究アプローチはアプリケーション以下のレイヤでシステム構成・環境の相違を吸収し、システム透過性の実現を目指すものが多い。これは有効なアプローチではあるが、少し視点を変えると多様な環境に対してアプリケーションをすべて同一に実行することが望ましいかどうかは疑問の余地が残る。つまりシステ

ムの構成や環境に応じてアプリケーションの振舞いを変えたほうが望ましい場合もあると思える。例えば、ネットワーク・アプリケーションの使用に際して、接続の媒体がイーサネットである場合と、電話回線、あるいは無線通信の場合では転送速度が大きく異なり、その実行効率に大きな差がでることは言うまでもない。このような環境では論理的なシステムの透過性を実現してもアプリケーションの効率的実行は望めない。そこでアプリケーションの振舞いをシステム構成や環境に応じて適宜切替えるというのが本研究のアプローチである。しかしこのアプローチはアプリケーションの実行速度の維持を主たる目的としているわけではない。アプリケーションの振舞いを変えたからといって、内在する物理的制約は解消されない。計算機の使用には、それにかかるコストが幾つか存在する。それが実行速度である場合もあれば、そうでない場合もある。例えば電話回線を用いたデータ転送では転送時間と転送にかかる金額というコストが考えられる。言うならば、これまでの計算機の使用は転送時間の短縮に主眼を置くものであった。しかし個人がそれぞれの目的でコンピュータを使用することが主体となる移動計算機環境では、実行速度のみが重視されるとは限らない。上記の例では転送にかかる金額も重要なコストとなる。つまり計算機の使用目的、使用形態に応じた処理を適宜選択し、実行するのが本研究の目的であり、それによりアプリケーションの効率的実行が可能になる場合もあるが、それのみを追求するものではない。

システム構成や環境に応じたアプリケーションの振舞いの実現には幾つかの方法が考えられる。まず、OS レベルあるいはそれに近いレイヤでこの種の機構を実現することが考えられるが、OS にかかる負荷の増大と言う点以外に、OS 本来の役割として、個々のアプリケーションの細かい振舞いまでサポートするというのは適当とは言えない。振舞いの変更はアプリケーションの性質、処理の性質に依存するものであり、全てのアプリケーションの振舞いに統一的にサポート可能なものではない。振舞いを変える主体はアプリケーションでなければならない。更に OS に組み込むことはカスタマイズを困難にする。このような機構を共有ライブラリとして実装し、アプリケーションを作成する際にそのライブラリを利用することも考えられる。しかし OS レベルでサポートする場合と同様、個々のアプリケーションの状況に適した振舞いを完全にサポートするのは不可能である。また各アプリケーション毎にそのような仕組みを内部に実装することも考えられる。しかしこれは実装にかかる負荷が大きいだけでなく、アプリケーションが本来目的とする処理コード中に、状況に依存する処理の制御コードが混在することになり、プログラムの可読性、保守性に問題が生じる。これは既存の言語がシステム構成や環境の変化に依存する部分と依存しない部分をモジュール化するように設計されていないからである。以上の点を考慮し、本研究では上記のモジュール化を促進する言語を新たに提案する。この言語の処理系はインタプリタとして実装されているが、言語処理系というよりはむしろアプリケーションの振舞いを状況に応じて選択し実行させるサーバとして捉えられる。通常のサーバと異なる点は、各アプリケーションに対して空の枠組を提供する点である。この空の枠組の中に各アプリケーションの処理を記述[1]し、それをシステム構成・環境に応じて選択し、アプリケーションに実行させるのである。つまり固定的な処理を提供せず、空の器を提供することで、静的な処理を行うサーバよりも多様

[1] 各アプリケーションが記述されている言語で記述する。

な環境に柔軟に対応可能となる。

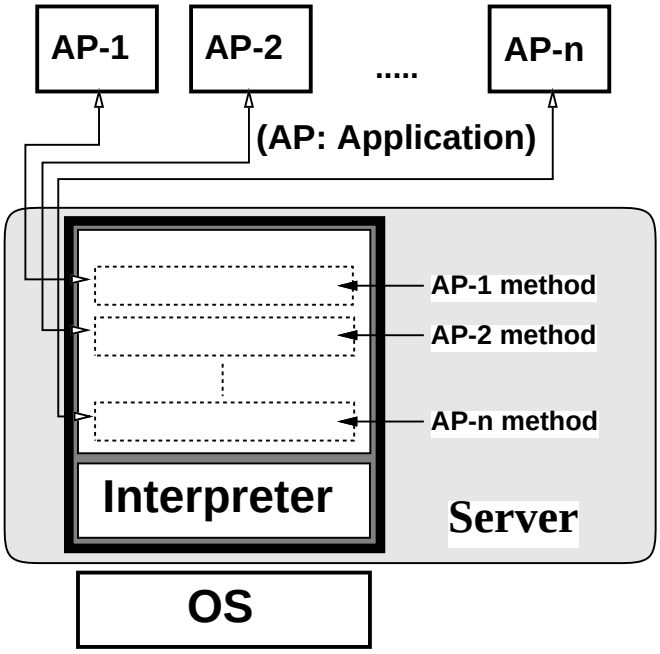

図 2: 本研究のアプローチ

1.3 本論文の構成

本論文は以下のように構成されている。まず第 2 章の各節において、本研究で提案する言語について、その機能及び実現を詳述し、第 3 章でこの言語を用いたアプリケーションについて述べる。また第 4 章では移動計算機環境に対する他の関連研究について触れ、第 5 章において結論を今後の課題を述べる。

2 プログラミング言語 LEAD

2.1 言語の概要

本研究で提案する言語を **LEAD**[2] と名付ける。LEAD は **Scheme** に範をとったインタプリタとして実現され、他のアプリケーションプログラムとやりとりするために、**socket** ベースのプロセス間通信機能を備えている。以下は Scheme をベースにした理由である。

1. プログラムをデータとして扱い易い。
2. 言語仕様が比較的コンパクトである。

上記 1 は Lisp 系言語に特徴的な点であるが、LEAD ではアプリケーションプログラムの一部分をデータとして扱うため、これは重要である。そして扱い易さ、及びその実

[2] The Language Describes AP's behavior adapts to the Environment. LEAD のプロトタイプは BSD/OS、SUN-OS 上に C 言語を用いて実装した。

装に際し、記憶容量等の資源が限定される移動計算機環境ではコンパクトであることが望ましい。またインタプリタ方式を採用した理由は以下による。

1. 実行系が必要である。
2. 環境に応じた動的なカスタマイズが可能になる。

動的なシステム構成に対応するには、常にアプリケーションやOSとやり取りする必要がある。つまり常に環境を認識できる実行系が必ず必要になる。またコンパイラ方式の場合、コンパイルされた情報は静的であり、動的に変更される可能性のある計算機システムに柔軟に対応することは困難である。そして利用者がインタラクティブにアプリケーションの動作を状況に応じて変更できる利点もある。ノート型パソコンやPDA端末のスペックは日進月歩の勢いで進歩しており、実行速度はインタプリタ方式でもあまり問題とならなくなりつつある。

LEADは言語として三つの側面を有する。各側面に便宜的に名称を付けることにする。

1. アプリケーション中のシステム構成・環境に依存する処理を記述する。(LEAD_f[3])
2. LEADインタプリタからアプリケーションを起動し実行させる処理を記述する。(LEAD_t[4])
3. システム構成・環境に依存する処理の制御（ディスパッチ）を記述する。(LEAD_d[5])

このようなLEADの側面とアプリケーションとの関係は以下のようになる。

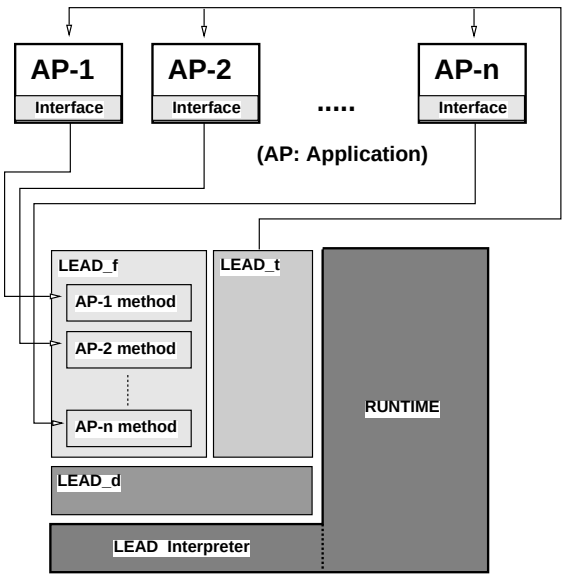

図 3: LEADとアプリケーション

[3] アプリケーションから(from)要求される処理。
[4] アプリケーションへ(to)要求する処理。
[5] 処理のディスパッチ(dispatch)。

つまり LEAD_f とはシステム構成や環境に応じたアプリケーションの異なる振舞いを記述するものであり、LEAD インタプリタ中に登録するものである。これらの振舞いは総称関数 (同一名称であり、かつ異なる振舞いをする処理 (メソッド) の総称) として定義される。各アプリケーションは LEAD インタプリタへのインターフェースを通して LEAD インタプリタに対し、総称関数の呼び出しを要求する。そして LEAD インタプリタがシステム構成・環境に応じたメソッドをアプリケーションに返し、各アプリケーション上でメソッドが実行される

次に LEAD_t とは LEAD インタプリタから能動的に各アプリケーションに働きかける処理を記述する。LEAD インタプリタはアプリケーションを起動し、先の LEAD_f として LEAD インタプリタ中に登録されているメソッドを目的のアプリケーションに送って実行させ、その結果を受け取る。LEAD_f と LEAD_t の大きく異なる点は前者がアプリケーションから要求される処理を記述するのに対して、後者は LEAD インタプリタからアプリケーションに要求する処理を記述するという点である。またこれは LEAD によって新規に作成されるアプリケーションも含む。

そして LEAD_d は LEAD インタプリタ中に登録された各アプリケーションのメソッドをシステム構成・環境に応じて選択するディスパッチの機構を記述するものであり、やはり LEAD によって記述されている。このメソッドのディスパッチは後述のポリシーに基づいて実現される。このディスパッチ・メカニズムが LEAD によって記述されているため、ディスパッチ・メカニズムの容易なカスタマイズが可能となる。これはシステム構成の変化しやすい計算機環境において特に有効であると考えられる。メソッドのディスパッチ・メカニズムは言わば LEAD のメタオブジェクトプロトコル [15] としてユーザーに公開されているものである。デフォルトのディスパッチポリシーが不適切であれば、より適切なポリシーにカスタマイズすることが可能である。

LEAD インタプリタがサポートするアプリケーションは以下の点を想定している。

- インタプリタ型の言語で記述されたアプリケーションであり、その実行系を持つ。
- アプリケーションを動かす実行系が socket ベースのプロセス間通信機能を備えている。

LEAD はそれ自体で一つの完結したアプリケーションを作成するだけの記述力を持った言語ではあるが、より効率的な使用方法として、他のインタプリタ言語で記述されたアプリケーションと組み合わせて用いることがあげられる。これにより以下の利点が得られる。

- アプリケーションをスクラッチから作る必要がなくなる。
- 既存のアプリケーションに対して、多様な環境に対応する機能を容易に付加することができる。

LEAD インタプリタとアプリケーションとの通信は **socket** を用いた双方向のプロセス間通信で実現されている。これによりアプリケーションあるいは LEAD インタプリタのいずれかがリモートにあっても容易に通信することができる。この通信を行なうため、アプリケーション側に LEAD との通信インターフェースが必要になる。本研究で

は **Emacs Lisp**、**Perl**、拡張 **Tcl/tk**[6]、**Python** の各言語処理系に対する LEAD へのインターフェース[7]を実装した。

2.2 LEAD の構造と動作

2.2.1 メタレベル ・アーキテクチャ

本研究で提案する LEAD とアプリケーションはメタレベル、ベースレベルの 2 層構造で表現され、メタレベル ・アーキテクチャとして構成される。各レベルは以下のように定義される。

- ベースレベル = アプリケーション
- メタレベル = LEAD インタプリタ

つまりベースレベルであるアプリケーション中の一部分 (システム構成や環境に応じて改変したい処理) を総称関数としてメタレベルである LEAD インタプリタ上で記述し分ける。より具体的に言えば、総称関数のうち異なる振舞いをする個々の処理をメソッドという単位で LEAD インタプリタに登録 (定義) する。LEAD インタプリタは、そのディスパッチ機構により、その時々のシステム構成や環境情報に基づいて、複数のメソッドから 1 つを選択し、そのメソッドをアプリケーションに返す。メソッドはそのアプリケーションが記述されているインタプリタ言語で記述されるが、LEAD 上では、それは単なる文字列データとして扱われる。このようなメタレベル ・アーキテクチャとして構成することでシステム構成や環境に応じたアプリケーションの異なる振舞いをサポートすることが可能になる。

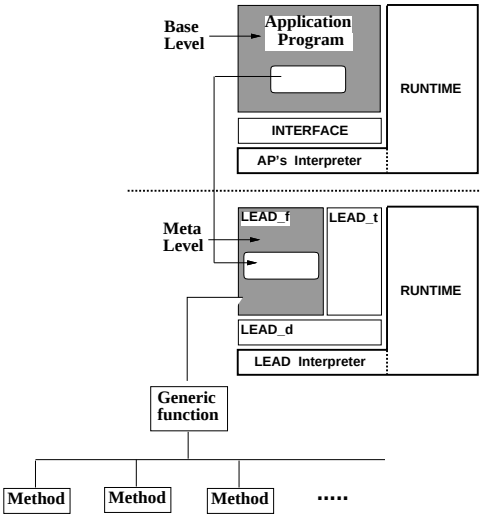

図 4: メタレベル ・アーキテクチャ

[6] Tcl/tk に socket ベースのプロセス間通信機能を持たせたもの。同様な機能を持つ Tcl-DP、Tcl-X 等もあるが、本研究では都合上自ら拡張したものを使用した。

[7] BSD/OS、SUN-OS 上に各アプリケーションが記述されている言語によって作成した。

2.2.2 アプリケーションの要求に基づく動作

ここで Emacs Lisp により記述されたメーラ (mailer) を例に LEAD をアプリケーションの動作を概観する。

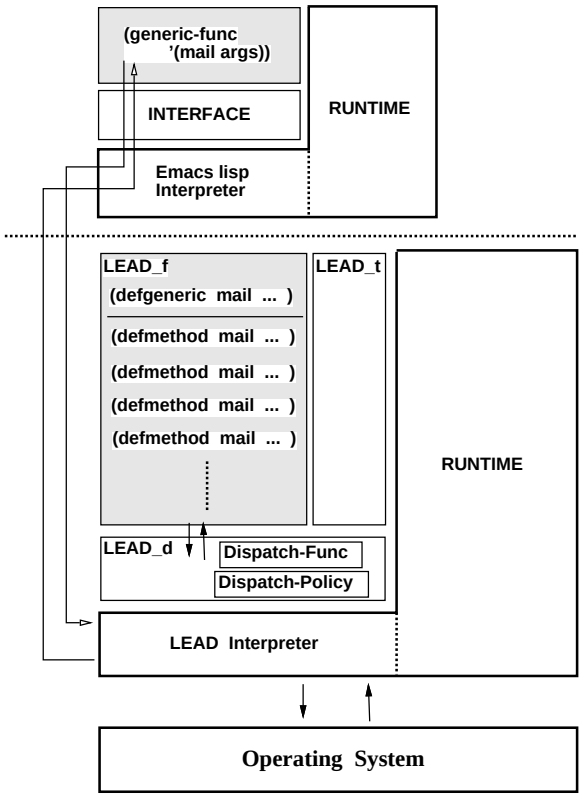

図 5: アプリケーションの要求に基づく動作

図中アプリケーションと OS の間に存在する一種のサーバが本研究で提案する LEAD インタプリタである。まず、メーラの中で状況に応じて振舞いを変えたい部分 (処理) を総称関数として LEAD インタプリタに定義し、異なる振舞いをする部分を個別にメソッドとして同様に定義する。これらの定義は図中に示す、LEAD の **defgeneric** 関数と **defmethod** 関数を用いて行なう。総称関数の定義では、その関数が必要とするシステムの状態を表す状態要素 (予約語) を列挙する。状態要素の幾つかは既にシステムの中に用意されている。そして個別のメソッド定義では、先に列挙した状態要素が示すシステムの状態についての条件を記述する。これがそのメソッドのディスパッチ条件となる。システムの状態に対する条件は複数指定することができる。またメソッドの処理部分はアプリケーションが記述されている言語によって記述する。これはその処理を実行する主体はアプリケーションを実行しているインタプリタを含めたアプリケーション側であるからである。メソッドは LEAD インタプリタ上のテーブル[8] にアプリケーション単位、総称関数単位に切り分けられて登録される。

アプリケーションは LEAD へのインターフェースを通じて LEAD インタプリタに対

[8] 実際はリスト。

し、総称関数の呼び出しを要求する。この時アプリケーションはLEADインタプリタとプロセス間通信を行ない、LEADインタプリタに対して現在の状況に適したメソッドをリクエストする。リクエストを受け取ったLEADインタプリタはメソッドが登録されているテーブルをサーチし、該当するアプリケーションのメソッドの集合を見つける。そして各メソッドのディスパッチ条件を評価し、ディスパッチポリシーに基づいてメソッドを特定する。(各メソッドのディスパッチ条件を評価する際にはOSから現在のシステム構成・環境情報を取得する。) 特定されたメソッドをプロセス間通信によってアプリケーションに送信する。アプリケーション側は受け取ったメソッドを評価(実行)する。

2.2.3 総称関数の定義と内部構造

総称関数はある処理を行なうメソッドの集合の総称であり、ある処理について個々のメソッドを記述する前に、その処理が総称関数であることを定義しなければならない。総称関数の定義は **defgeneric** 関数を用いて以下のように行う。

```
(defgeneric 'method-name
            'app-lang
            '(stat-1 ... stat-n))
 method-name : 総称関数名
 app-lang    : アプリケーション記述言語名
 stat-1 ... stat-n : 状態要素のリスト
```

状態要素とは環境や状況に応じて変化する可能性のあるデバイス等を指す。例えばメモリ、バッテリ等がある。これらの状態要素の幾つかは予めシステム中に定義されているが、ユーザーが任意に定義することも、再定義も可能となっている。総称関数で使用される状態要素はその優先度が高い順に左から指定する。そしてこの関数の実行により、状態要素の優先度を示す優先度数リストが生成される。

【例】

```
(defgeneric 'func
            'emacs
            '(network memory battery))
 funcemacs : ((network . 8)
              (memory . 4)
              (battery . 2))
```

上記の例の実行により、総称関数名とアプリケーション記述言語名からなるfuncemacsというシンボル[9]に生成された属性リストがバインドされる。つまりこのような属性リストが各アプリケーションに対する総称関数毎に生成される。各状態要素とペアになっている数値はその状態要素の優先度数を表す。この優先度数は指定された状態要素の数をnとすると2^n−状態要素の優先順位で算出される。上記の例であげた状態要素

```
network, memory, battery, display
```

[9] これも実行時に生成される。

等は予めシステム中に定義されているものである[10]。これらは各要素の状態を調べる関数と結び付けられており、各要素の状態を調べ、その状態を値として返す、無引数の関数である。状態要素を括弧でくくることによりその要素の状態を示す関数呼び出しとなる。

(network) : ネットワークの種別を返す関数。
(memory) : メモリの容量を返す関数。
(battery) : バッテリの残量を返す関数。
(display) : ディスプレイサイズを返す関数。

このように状態要素はこの言語の関数として定義されている。これはとりもなおさず、ユーザーが定義した関数を状態要素として用いることができるという容易な拡張性をもたらす。予め用意された状態要素には必然的に限りがある。しかし、システム構成・環境が動的に変化する可能性のある環境に、限られた状態要素で対応することは困難であると思われる。勿論 OS レベルでサポートされていないような機能を付加することは不可能であるが、予め定義されているこれらの関数を組み合わせることにより、新たな状態要素を定義することも可能になる。例えば、ネットワークの種別 (network) やバンド幅の状態 (band-width) 等の情報から、データ転送にかかる money(金額) という状態要素を新たに定義することが可能になる[11]。

【状態要素 money の追加定義】

```
(define money
  (lambda ()
    (let ((net (network))
          (band (band-width)))
      (begin
        (       ...     )))))
```

(money) : 転送にかかる金額を返す関数呼び出し。
(返り値は K バイト単位にかかる金額等)

2.2.4 メソッドの定義と内部構造

メソッドは個々のアプリケーション中の状況に依存した振舞いであり、**defmethod** 関数を用いて以下のように定義する。

```
(defmethod 'method-name
           'app-lang
           '(cond-1 ... cond-n)
           "host-name"
           "source-code")
```

method-name : メソッド名
app-lang : アプリケーション記述言語名

[10] 本研究では BSD/OS、SUN-OS 上にこれらを実装した。
[11] このような状態要素を新たに追加することができるように、システムの状態を取得する関数を複数サポートする予定である。

cond-1 ... cond-n : ディスパッチ条件リスト
host-name : 実行ホスト名
source-code : 実行ソースコード

defmethod 関数の実行により生成されたデータはメソッドを登録するテーブルにアプリケーション単位、総称関数単位に区分けされ登録される。このテーブルは大域シンボル func-array にバインドされる。メソッドのディスパッチはこのテーブルを参照することによりなされる。func-array の初期値は空リストである。

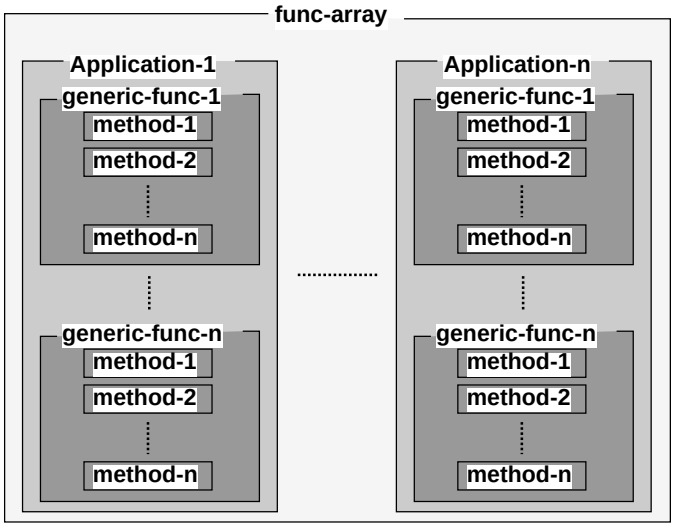

図 6: 大域シンボル func-array

ディスパッチ条件リストとはメソッドが選択される条件を示したリストであり、defgeneric 関数で指定した状態要素に対する条件を指定したリストである。つまり各状態要素の状態をディスパッチの条件として記述するのである。この条件の記述方法は、その状態要素が関連付けられている関数の返す値に依存しているが、条件演算子は、中置記法を用いる。そして、ここで使用可能な状態要素はこのメソッドに対する状態要素として defgeneric 関数で定義した状態要素に限られることである。また複数の条件を記述することも可能であるが、それらの条件は and 条件でも or 条件でもない。これについては後述する。

次に実行ホスト名とは続く実行ソースコードを実際に実行するホストを指定する。これはこの言語の機能として特徴的な状況に応じたプログラムのリモート実行を行なうために必要となる。この実行ホスト名には **local** と **remote** という 2 つの予約語がある。local はローカルホストを、remote はデフォルトのリモートホストをそれぞれ指す。それ以外のリモートホストはそのホスト名を直に記述する。

また実行ソースコードとはアプリケーション側に送られ実行されるソースコードであり、アプリケーション記述言語として指定された言語によって記述されるものである。実例に即して defmethod 関数の実行により生成される内部構造をみる。

【例】

```
(defmethod 'test
           'emacs
           '((network = 1)
             (memory >= 10)
             (battery >= 50))
           "local"
           "(defun test (n) (* n n))" )
```

この例は Emacs で記述されたアプリケーション中の test という関数をメソッドとして定義している。実行はローカルホストで行い、実際の処理 (Emacs 上で実行される関数。) は

```
"(defun test (n) (* n n))"
```

である。このメソッドがディスパッチされる条件は、

```
'((network = 1)
  (memory >= 10)
  (battery >= 50))
```

である。この例の実行で次のリストが生成される。

```
((emacs
  ((test
    ((((network = 1)
       (memory >= 10)
       (battery >= 50)) .
    "local (defun test (n) (* n n))"))))))
```

このようなメソッド及び総称関数を LEAD インタプリタに定義する言語としての側面が LEAD における LEAD_f である。

2.2.5 メソッドのディスパッチ

LEAD におけるメソッドのディスパッチ・アルゴリズムは前述した defgeneric 関数によって生成された状態要素の優先度数リストと、defmethod 関数によって生成されたリスト (以下、メソッドリスト) を参照・評価することで実現される。このメソッドのディスパッチ・アルゴリズムは LEAD によって実装されている。これは先に述べた LEAD の一つの側面 LEAD_d を指す。

あるアプリケーションからメソッドの呼び出し要求が出されると、LEAD インタプリタは受け取ったアプリケーション記述言語名とメソッド名をキーとして大域シンボル func-array の中からメソッドを特定する。その際にメソッドリスト中のディスパッチ条件リストの各条件が評価され、現在のシステム構成に適したメソッドが選択される。実例に即して動作をみる。

【例】 この例において以下の条件を仮定する。

1. アプリケーション: Emacs
2. 要求されるメソッド: test
3. 大域シンボル func-array:

```
((emacs
  ((test
    ((((memory < 10)
       (network = 1)) .
    "local (defun test (n) (* n n))")
     (((battery < 20)
       (network = 0)
       (display = 1)) .
    "local (defun test (n) (* n 5))")))
   (func ... ))))
 (perl
  ((test ... )))))
```

4. 状態要素の優先度数リスト testemacs:

```
             ((memory . 8)
              (battery . 4)
              (network . 2)
              (display . 1))
```

5. memory の現在値 : 7 (MB)
6. battery の現在値 : 15 (%)
7. network の現在値 : 0 (= LAN)
8. display の現在値 : 1 (1159 : 900)

アプリケーション (Emacs) から LEAD に渡されるメッセージはアプリケーション名 emacs と要求するメソッド名 test である。以下がディスパッチの動作である。

(1) アプリケーション名 emacs より、大域シンボル func-array 中の以下の部分に特定できる。

```
(emacs
 ((test
   ((((memory < 10)
      (network = 1)) .
   "local (defun test (n) (* n n))")
    (((battery < 20)
      (network = 0)
      (display = 1)) .
    "local (defun test (n) (* n 5))")))
  (func ... ))))
```

(2) 次にメソッド名 test より (1) のリスト中の以下の部分に特定される。

```
(test
 ((((memory < 10)
    (network = 1)) .
 "local (defun test (n) (* n n))")
  (((battery < 20)
    (network = 0)
    (display = 1)) .
 "local (defun test (n) (* n 5))")))
```

(3) (2) のリストの cdr 部をとる。

```
((((memory < 10)
   (network = 1)) .
   "local (defun test (n) (* n n))")
 (((battery < 20)
   (network = 0)
   (display = 1)) .
   "local (defun test (n) (* n 5))"))
```

(4) (3) で抽出したリストの car 部をとる。

```
(((memory < 10) (network = 1)) .
  "local (defun test (n) (* n n))")
```

(5) (4) で抽出したリストの car 部をとる。

```
((memory < 10) (network = 1))
```

(6) (5) のリストの要素をそれぞれ評価する。

(6-1) まず (memory $<$ 10) を評価する。この条件式は内部で以下のように変換され、評価される。

```
(memory < 10)   = (< (memory) 10)
(memory)        = 7 (MB)   (仮定)
(< (memory) 10) = (< 7 10)
(< 7 10)        = true
```

よって (memory $<$ 10) の評価値は真。

(6-2) 次に状態要素の優先度数リストを参照し、状態要素に対する条件が真の場合はその優先度数を一時変数 (true-point) に加算していき、偽の場合はその優先度数に-1 掛けた値を、別の一時変数 (false-point) に加算していく。[12]

[12] true-point, false-point の初期値はともに 0 である。

memory の優先度数 = 8　(仮定)

true-point = 8

(6-3) 上記 (5) で得たリストの要素全てに対して (6) の操作を行なう。

```
(network = 1)   = (= (network) 1)
(network)       =  0   (仮定)
(= (network) 1) = (= 0 1)
(= 0 1)         =  false
```

network の優先度数 = 2　(仮定)

false-point = -2

以上から上記 (4) で抽出されたリスト

```
(((memory < 10) (network = 1)) .
  "local (defun test (n) (* n n))")
```

に対して、

```
true-point  =  8
false-point = -2
```

という評価値が確定されたことになる。この操作を上記 (3) で抽出されたリストの全ての要素に適用する。すると

```
(((battery < 20)
  (network = 0)
  (display = 1)) .
  "local (defun test (n) (* n 5))")
```

という (3) の cadr 部のメソッドに対して、

```
true-point  = 7
false-point = 0
```

という評価値が確定する。

(7) 以上でメソッド名 test に対する全てのディスパッチ条件が評価され、次の結果を得た。

```
(A) true-point = 8 , false-point = -2
    "local (defun test (n) (* n n))"
(B) true-point = 7 , false-point =  0
    "local (defun test (n) (* n 5))"
```

この二者のうちどちらを選択するかがディスパッチのポリシーである。LEAD におけるメソッド選択のデフォルトポリシーは以下の二つのルールに基づいている。

(a) false-point がより大きいものを優先する。

(b) true-point がより大きいものを優先する。

そして (a) は (b) よりも優先される。つまり false-point が同値であるメソッドに対して、(b) を適用することでメソッドの特定化を行なう。先の例でメソッド (A)、(B) の false-point を比較すると、(B) の方が大きいので (B) のメソッドが選択され、

```
"local (defun test (n) (* n 5))"
```

上記の文字列が、要求されたメソッドとしてアプリケーションに送られる。

2.2.6 ディスパッチのポリシー

先に述べたように LEAD のデフォルトのディスパッチポリシーは以下のルールで実現されている。

(a) 各状態要素に対する条件が偽となるものの合計値 (false-point) が最大であるものを優先する。

(b) 各状態要素に対する条件が真となるものの合計値 (true-point) が最大であるものを優先する。

優先順位は **(a)** > **(b)** で、**(a)** は **(b)** よりも優先される。このようなディスパッチのポリシーを選択した理由について、先に用いた例を交えて述べる。下記は先に用いた例であり、このリストは test というメソッド名をもつメソッドの集合である。この例ではメソッドの集合は以下の二つのメソッドに分けられる。

```
((((memory < 10)
   (network = 1)) .
   "local (defun test (n) (* n n))")
 (((battery < 20)
   (network = 0)
   (display = 1)) .
   "local (defun test (n) (* n 5))"))
--------------------------------------
(A) (((memory < 10)
      (network = 1)) .
      "local (defun test (n) (* n n))")
(B) (((battery < 20)
      (network = 0)
      (display = 1)) .
      "local (defun test (n) (* n 5))")
```

メソッド (A) のディスパッチ条件リストは、

```
((memory < 10)
 (network = 1))
```

メソッド (B) のディスパッチ条件リストは、

```
((battery < 20)
 (network = 0)
 (display = 1))
```

となっている。仮定よりそれぞれの条件式の評価結果は、

```
(A) = (( 真 ) ( 偽 ))
(B) = (( 真 ) ( 真 ) ( 真 ))
```

であった。これをみると (A) は条件のうち一つが偽となっているのに対して、(B) は全ての条件が満たされている。つまり指定されたディスパッチ条件を全て満たしている (B) のメソッドを選択することは自然である。少なくとも指定された条件式が偽となる数の少ないメソッドがそうでないメソッドより優先されることは理に適っている。しかし LEAD のポリシーは真や偽となる条件式の数の大小のみによるものではない。なぜならば、そのような単純なポリシーでは対応できないケースが存在するからである。例えば、真偽の数が等しいような次の二つのメソッドのケースを考える。

```
(C) (((memory < 10) (network = 1)) .
      "local (defun test (n) (* n n))")
(D) (((battery < 20) (display = 1)) .
      "local (defun test (n) (* n 5))")
```

上記のメソッドのディスパッチ条件リストはそれぞれ

```
(C) ((memory < 10) (network = 1))
(D) ((battery < 20) (display = 1))
```

である。今 (C)、(D) の評価結果がいずれも

```
(C) = (( 真 ) ( 偽 ))
(D) = (( 真 ) ( 偽 ))
```

であった場合、どちらのメソッドを選択するべきか ?ここで状態要素間の優先順位が必要となる。つまり真や偽となる条件式の数だけではメソッドを特定することが不可能であり、それを回避するために状態要素間の優先順位を設けている。ここで上記のメソッドに対する優先度数リストを

```
((memory . 8)
 (battery . 4)
 (network . 2)
 (display . 1))
```

とすると、

```
(C) true-point  =  8
    false-point = -2
(D) true-point  =  4
    false-point = -1
```

となり、false-point がより大きい (D) が (C) に優先されて選択される。ここには優先度数の高い状態要素の条件が満たされなかったメソッドほど優先されない。というポリシーが働いている。ユーザーは defgeneric 関数を用いて、あるメソッドに対する状態要素の優先度数を決定する。つまり優先したい状態要素をユーザー自らが指定しているに他ならない。これは前述したように、最も優先したい状態要素を左から順に書き並べることで実現される。ここで優先という語は重視と置き換えることができる。すなわち

```
(defgeneric 'method-name
            'app-lang
            '((1) (2) (3) (4)))
```

のような例において、最左の状態要素 (1) はその他の状態要素よりもユーザーが重視したい項目と考えられる。このメソッドについて状態要素 (1) が偽となる場合と状態要素 (4) が偽となる場合の異なる二つのメソッドがあった場合、(1) が偽となる方を選択するか、それとも (4) が偽となる方を選択するか、が問題となる。この場合、先の例でも分かるように優先度数の低い (4) が偽となる方が選択される。それは重視している条件が不成立のものよりも、あまり重要でない条件が不成立のものを選択する方が妥当であるからである。これは先にあげた LEAD のディスパッチポリシー (a) に相当する。

次に先の例を用いて (C)、(D) の評価結果がいずれも

```
(C) = (( 真 ) ( 真 ))
(D) = (( 真 ) ( 真 ))
```

となるケースを考える。この時、やはり先ほどと同じ条件を仮定すると

```
(C) true-point  = 10
    false-point =  0
(D) true-point  =  5
    false-point =  0
```

となる。(C)、(D) ともに false-point は 0 である。このような時に優先度数の高い状態要素の条件が満たされたメソッドほど優先される。というポリシーが働く。これは先に述べた優先度数の高い状態要素の条件が満たされなかったメソッドが優先されないことの逆である。より重視している条件が満たされたケースの方が、重要でない条件を満たしたケースよりも優先されることは自然である。これはやはり前述した LEAD のディスパッチポリシー (b) に相当する。以上が LEAD におけるメソッドのディスパッチポリシーである。

2.2.7 状態要素の追加と削除

defgeneric 関数で指定する状態要素の数を変更する (追加や削除) と、各状態要素間の優先度数が変化する。注意すべきことは、各メソッドのディスパッチ条件で使用する状態要素は全て記述しなければならないことである。あるメソッドについて状態要素を新たに追加したい場合は、再度 defgeneric 関数を用いて、使用可能な状態要素を定義し直さなければならない。この際 defmethod 関数を新たに実行し直す必要はないが、場合

によってはその必要がある時もある。つまり新たな状態要素の追加にともない、状態要素の優先度数が変化するため、古い優先度数下で考慮されたメソッドが新しい優先度数下で今までと同様にディスパッチされる保証はないからである。これは状態要素を削除した場合に、より問題となる。削除した状態要素を使用しているメソッドは評価時にエラーとなる。このように既にあるメソッドの状態要素に対する追加や削除は十分な注意が必要である。

2.2.8 LEAD によるアプリケーションの実行

LEAD インタプリタからアプリケーションを実行する機能は LEAD_t と名付けた LEAD の一つの側面を指す。この LEAD_t は複数のアプリケーション間に渡る振舞いを状況に応じて改変するシェルスクリプトと捉えられる。つまりシステム構成・環境に応じてアプリケーションを切替えたりする処理を記述する。よく使用する複数のアプリケーションを用いた一連の処理をシェルスクリプトとして定義することは一般的である。UNIX 上のシェルは制御構造をサポートしているので、一連の処理を分岐させたり、繰り返したりすることは可能であるし、またある環境変数等を参照し、ある程度環境に依存した振舞いをするシェルスクリプトも実現できる。あるいは Perl などと組合せれば、より複雑な環境に依存した振舞いのシェルスクリプトを実現できるかもしれない。しかしこれらのシェルスクリプトでは限界がある。つまり各アプリケーションの中の処理までシステム構成・環境に依存した振舞いを実現させることはできない。つまり LEAD_t によるスクリプトは通常のシェルスクリプトよりも一段深いレベルの環境に依存した振舞いを実現可能にする。それは LEAD インタプリタからアプリケーションを実行する際、アプリケーションに要求する処理は LEAD インタプリタにメソッドとして定義されているものでなければならないからである。すなわち、状況に応じたアプリケーションの切替えと同時に各アプリケーション中の振舞いも状況に応じて改変する。このようなスクリプトを LEAD では特別に区別することなく通常の関数と同様に記述することができる。これは先に述べた状態要素が、通常の関数同様いつでも呼び出すことができるからである。これを用いてシステム構成・環境に応じてアプリケーションを切替えることが可能となる。アプリケーションに対する実行要求は次のように記述する。

```
(call ' アプリケーション名 "関数呼び出し")
```

【例】

```
(call 'perl "&test(80)")
```

上記の例は、Perl に対して test という総称関数の実行を引数 80 で依頼し、結果を値として受け取る、ということを示している。なお test という関数が総称関数として LEAD インタプリタに定義されていることを前提とする。つまり call 関数は次のように動作する。

1. 引数からアプリケーションと総称関数名を特定する。
2. 大域シンボル func-array (総称関数テーブル) をみてメソッドの集合を取得する。
3. 個々のメソッドを評価し、現在のシステム構成・環境に適したメソッドを選択する。

4. 選択したメソッドと引数として受け取った関数呼び出しをセットにして一時ファイルを作成する。

5. 作成した一時ファイルを引数として該当アプリケーションを起動する。

6. アプリケーションが終了するまで LEAD は待ち状態となる。

7. アプリケーションの返り値を標準入力を通じて取得する。

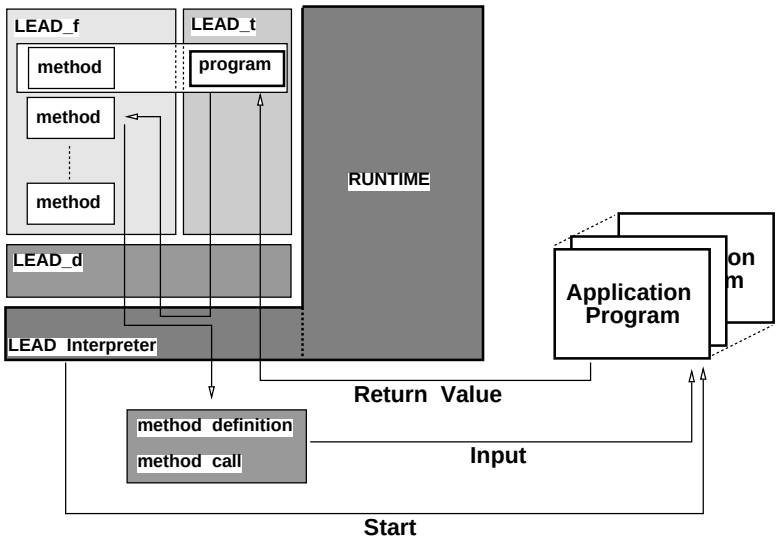

図 7: LEAD によるアプリケーションの実行

2.3 アプリケーションの効率的実行

LEAD はアプリケーションの実行効率を維持するために以下の機能をサポートする。

- プログラムのリモート実行
- QoS(Quality of Service)

これらの機能をサポートする理由は、移動計算機環境では通常の作業環境としてのデスクトップ環境と同等のレスポンスを維持したいという要求も常に存在するからである。アプリケーションの実行効率という点に関して上記の機能は有効である。これらの機能は本研究のアプローチを越え出るものではなく、包含されるものである。つまりシステム構成や環境に応じてアプリケーションの振舞いを変化させることのなかには、このように機能するものも含まれる。例えばホストマシンの CPU 性能に応じてプログラムをリモート実行する等の振舞いはアプリケーションの実行効率に貢献する。上記の機能によりシステム構成 ・環境に依存しないアプリケーションの効率的実行が可能となる。

2.3.1 プログラムのリモート実行

システム構成や環境の動的変化に対し、アプリケーションの実行効率を考えた場合、プログラムのリモート実行が有効な手段と成り得る。これは移動計算機環境が内包する以下の物理的問題に関連する。

- 容量的な問題
  - ・メモリ容量
  - ・ハードディスク容量
- ネットワーク上の問題
  - ・狭いバンド幅
  - ・一時的な回線切断の多発
- CPU 性能の問題

つまりこのような物理的制約があるため、これらの制約がない環境でのアプリケーションの実行と比較した時、同じアプリケーションの実行でも効率上の差がでることは容易に想像できる。プログラムのリモート実行は異なる環境におけるアプリケーションの実行効率上の差の解消を意図している。つまり PDA のような携帯端末でデスクトップ環境と同等なアプリケーションの実行効率を目指した場合、高い処理能力を持つリモートホストの CPU パワーを得ることでレスポンスを維持することができる。また LEAD におけるプログラムのリモート実行は、既存の **RPC** にない以下の利点を有する。

1. システム構成 ・環境に応じて動的にリモート実行を行なわせる。
2. プログラム自体を送信するため、リモートホスト上にないプログラムの実行が可能である。

2.3.2 QoS(Quality of Service)

プログラムのリモート実行と対にして考えられるものが QoS の機能である。**QoS** [5, 10] とはシステム構成 ・環境に応じて動的にデータ等の質を調整し、サービス全体としての質を保証することを意味する。プログラムのリモート実行はアプリケーションの実行効率を維持する有効な手段であるが、負荷のかかる処理をリモートで実行することができても、その結果の表示はローカルホスト上 (ここではノート型パソコンや PDA 等の移動計算機を指す。) で行なわなければならない。つまり移動計算機側で結果を表示する際に負荷がかかる場合、例えば結果の表示がフルカラーの画像データなどであった場合、QoS はその負荷を軽減する有効な手段と成り得る。この例のような場合、フルカラーの画像を白黒画像におとす QoS が考えられる。これによって表示データのサイズを抑え、表示に伴う負荷を軽減し、極度のレスポンスの低下を招くことなくアプリケーションの実行が可能になる。この QoS に関して本研究では一切実装しておらず、将来的にサポートする予定である。

3 例題 :テキスト ・エディタ

3.1 仕様と振舞い

本研究では例題として LEAD と拡張 Tcl/tk を用いて、システム構成 ・環境に応じて振舞いを変えるテキスト ・エディタを作成した。このテキスト ・エディタ[13] は GUI ベースで実現されるフルスクリーン ・エディタであり、システム構成や環境に依存した振舞いの仕様は以下である。

1. ディスプレイのサイズに依存した振舞い
   - メインウインドウのリサイズ
   - 文字フォントのリサイズ
   - ダイアログのリサイズ ・Widget の配置替え
2. メモリ容量とバッテリ残量に依存した振舞い
   - 文字入出力バッファの伸縮
3. バッテリ残量に依存した振舞い
   - ファイルの自動セーブ回数の増減

このような仕様のテキスト ・エディタの振舞いについて順に述べる。

1. 使用するディスプレイのサイズに合わせてアプリケーションのウインドウをリサイズすることは適切な処理である。例えば 9 インチディスプレイのノート型パソコンを使用している時に画面からはみ出る程、大きなウインドウを開いても操作しにくい上、他のウインドウを開くことができなくなる。また逆に 21 インチディスプレイを使用しているような時は、それなりに大きいウインドウにしたい。見易さという点からそれは望ましいことと言える。そしてこれはメインウインドウ以外の一時的なダイアログウインドウにも当てはまることである。 (ここではディスプレイサイズに応じたウインドウの最適な大きさと言う点については議論しない。) 同様に文字のフォントサイズについてもある程度同じことが言える。

   今まで述べたウインドウや文字フォントのリサイズに関しては LEAD のような言語を使用しなくてもリソースファイルに記述することで対応可能である。しかしリソースファイルで記述できるものはその範囲が極めて限定されている。メインウインドウやデフォルト文字サイズ程度ならば、リソースファイルで対応可能であるが、アプリケーション内のダイアログウインドウ等について、あるいは別の処理に関してリソースファイルで制御することは出来ない。それはリソースファイルがあくまでもアプリケーションのリソースの属性についての記述ファイルであり、処理を記述するものではないからである。

[13] このテキスト ・エディタは参考文献 [8] のサンプルをもとに BSD/OS、SUN-OS 上に作成した。

LEAD ではこれらの処理が動的に変更可能となる。(アプリケーション起動時に決まるものについては再起動する必要がある。) 例えばダイアログウインドウ等については再起動することなく、変更を動的に反映させることができる。

またディスプレイサイズに依存する処理として Widget の配置替えも行う。Widget とは GUI ベースのアプリケーションを構成する部品である。具体例を示すと、下図はこのエディタで編集するファイルを選択する時に開くダイアログウインドウである。

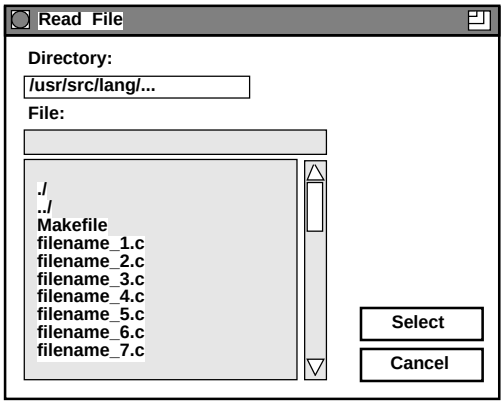

図 8: ファイル選択ダイアログウインドウ (1)

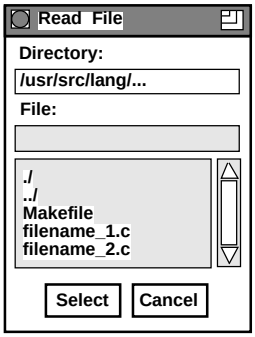

図 9: ファイル選択ダイアログウインドウ (2)

ディスプレイの大きさに依存して図のように 2 通りのダイアログウインドウが表示される。(1) はサイズの大きいディスプレイの場合で、(2) はサイズの小さいディスプレイの時、開かれる。以下のことが直観的に明らかである。

- ディスプレイのサイズに応じて Widget のサイズが異なる。
- ディスプレイのサイズに応じて Widget の配置が異なる。

問題は Widget のリサイズよりも配置である。Widget の配置替えが、ダイアログウインドウ全体に与える影響は大きい。こうした Widget の配置は各アプリケー

ションに依存する性格が強いと言える。つまり、これと同様の処理を汎用のプラットフォームを用いて均一に実現した場合、問題が生じるケースがある。Widget の配置に重要な意味を持たせているようなアプリケーションでは、その配置を替えたくないかもしれない。配置を替えてもいいアプリケーション、替えたくないアプリケーションに対して汎用のプラットフォームで対応するのは困難である。アプリケーションに依存した処理はアプリケーションに任せるのが最も望ましい。

2. メモリの大きさに依存してアプリケーションの振舞いを変えることも妥当な処理である。これは特にメモリ容量が制限される PDA 等では有効な手段と考えられる。メモリの総容量が少なければメモリを少しでも節約することが望まれる。文字入出力バッファはディスクアクセスを減らし、効率的な入出力を実現する。入出力バッファのサイズを大きく取れば、ディスクアクセスの回数を減少し、効率的な入出力を実現するだけでなく、ディスクアクセスに伴う消費電力の節減にも有効である。しかし、メモリが少ない場合は多少ディスクアクセスの回数が増えても、入出力バッファのサイズを抑えて、メモリの消費を節減したい。メモリとバッテリのどちらを優先するかは、そのアプリケーションの性質と処理の性質による。ここではメモリの消費を優先している。

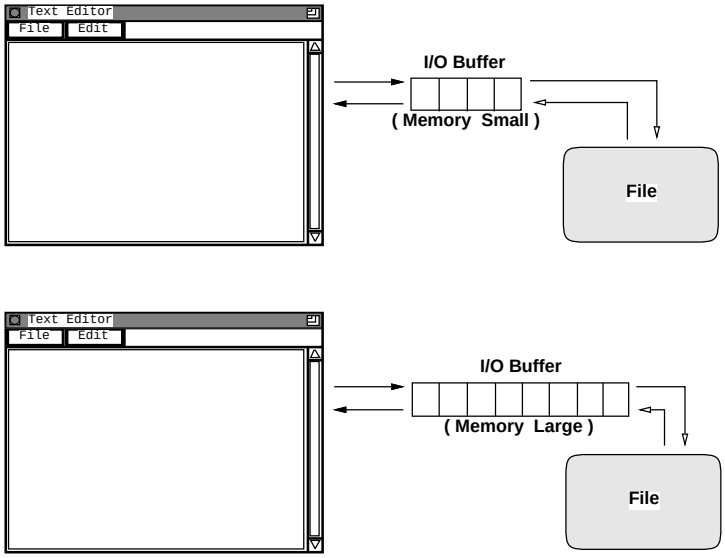

図 10: 文字入出力バッファの伸縮

3. このエディタはファイルの自動セーブ機能を持つ。動作の概略を述べると編集した文字数によって一時ファイルに自動セーブを行なうメカニズムとなっている。移動計算機環境のような電源の確保が難しい状況も有り得る環境ではこの機能の必要性は十分に考えられる。つまりバッテリで駆動している際、ファイルの編集途中でバッテリが切れる可能性があるからである。ここではバッテリが重要な要素となる。すなわちバッテリの残量が少ない時には自動セーブの回数を抑えてディスクアクセスを減らし、バッテリの消耗を回避する。

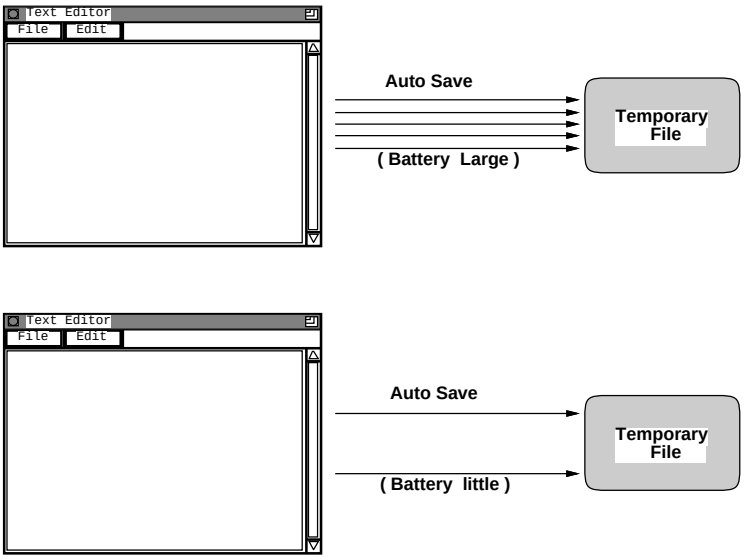

図 11: ファイル自動セーブの回数の増減

3.2 評価

このテキスト・エディタの評価には二つの難しさがある。一つは数値として表せないものの場合である。例えばウインドウのリサイズ、Widget の配置替え等について、その有効性を数値で示すことはできない。また物理的な容量に伴う問題の場合、その節減の有効性については動作環境によって結果が異なる可能性が大きい。このため本研究では厳密な評価を行なってはいない。しかし次のことを LEAD の有効性としてあげることができる。つまり、このようなシステム構成・環境に依存した振舞いをアプリケーションに付加するのにかかる労力は **LEAD** を用いることによってかなり軽減されるということである。それはエディタの作成にかかったコーディングの量を比較すれば一目瞭然である。

拡張 Tcl/Tk のコード = 約 1200 ステップ
LEAD のコード = 総称関数 :6 メソッド :14

拡張 Tcl/tk の約 1200 ステップに対して LEAD の記述量はごくわずかである[14]。しかも LEAD のコードはその大半が拡張 Tcl/tk のコードであるといって差し支えない[15]。このようなごくわずかの記述量で既存のアプリケーションに対して、システム構成や環境に動的に対応できる機能を付加することができる。

[14] LEAD のコーディング量をステップで算出していないのは見易すさの為の改行を行っているからである。
[15] 実行ソースコードの部分が最もコーディング量が多いため。

4 関連研究

移動計算機環境に対するアプローチの多くはOS レベルもしくはそれに近いレベルでシステムの透過性を実現するものであり、本研究のアプローチとは大きく異なる。例えばホスト移動にともなうネットワーク透過性の実現にはVIP 方式 [14]、Host Route 方式、Source Routing 方式等が提案されている。また動的なシステム構成の変化に対応すると言う観点ではマイクロカーネルベースのOS に機能拡張するものが多く見られる。それらの例としてCache Kernel [16] やFlux [17] 等がある。これ以外にもアプリケーションに対して実行時にカーネル内にコードのロードを許可するSPIN [18] 等、Adaptive なOS が研究、提案されている。

5 結論

以上、本研究においてメタレベル・アーキテクチャを用いた移動計算機環境を支援する言語処理系LEAD を提案し、その構造と動作について述べた。このようなLEAD の有効性については以下のようにまとめることができる。

- システム構成・環境の動的変化に対応する部分をアプリケーション本体から分離(モジュール化)して記述できる。(保守性・可読性を高める。)
- システム構成・環境の動的変化に応じたアプリケーションの振舞いを選択する共通の機構をアプリケーションに対して提供する。
- 既存の言語からなるアプリケーションに、容易にシステム構成や環境に依存した振舞いを実現する機能を付加することができる。
- メソッドの実行ソースコード部分の記述は特定の言語に限定されない。(アプリケーションの振舞いはそのアプリケーションが記述されている言語によって記述することができる。)
- メタオブジェクトプロトコルとしてメソッド選択のポリシーをユーザーに公開しているので、システム構成・環境に応じてこのポリシーを柔軟にカスタマイズすることができる。
- LEAD インタプリタ上にダミーの状態要素を作成することであるシステムの状態を容易かつ動的にシミュレートすることが可能となる。
- プログラムのリモート実行やQoS を行なうことでシステム構成・環境によらずアプリケーションの実行効率を維持することができる。

また今後の課題として以下の点があげられる。

- アプリケーションとの非同期通信の実現
- LEAD_t の完全実装
- リモート実行機能の完全実装

- QoS の実装
- RT-Mach 上への LEAD の実装 (移植)
- アプリケーション の有効性の検証

本研究における実装では LEAD インタプリタは他のアプリケーションと 1 対 1 の同期通信を行なう。しかし移動計算機環境のような環境では非同期に通信できることが望ましい。特に LEAD_t のような機能は複数のアプリケーションとやり取りするスクリプトの記述であり、この場合複数のアプリケーションと同時に通信できることが要求される。例えばあるアプリケーションが負荷のかかるものであった場合、その処理とは直接関係ない他のアプリケーションの処理が実行できる方がより望ましい。

また LEAD は **RT-Mach** [13] 上に実装 (移植) を予定している。RT-Mach は以下のような特徴を持った OS である。

- マイクロカーネルに基づく比較的コンパクトな OS である。
- 計算機構成や環境の動的変化に対応し易いモジュール構造を持つ。

このような特徴を持つ RT-Mach は従来の UNIX などと比較して移動計算機環境に適した OS であると言える。こうした RT-Mach 上で LEAD を用いたアプリケーションの有効性を検証することで、LEAD の有効性をあらためて検証する。

謝辞

この研究は情報処理振興事業協会 (IPA) が実施している独創的情報技術育成事業 「移動計算機環境におけるアプリケーション構築法に関する研究」プロジェクトのサポートを受けている。

参考文献

[1] 天野 憲樹. Meta-level Architecture を用いた移動計算機環境を支援する言語処理系の実現, 北陸先端科学技術大学院大学, 修士論文, February 1995.

[2] G.H.Forman and J.Zahorjan. The Challenges of Mobile Computing, IEEE Computer, Vol.27, No.4, 1994.

[3] 保木本 晃弘, 中島 達夫. オブジェクト指向に基づくモーバイルコンピューティングに適したカスタマイズ可能なＯＳ構築のための枠組, In Proceeding of the 10th Workshop of Object-Oriented Computing, 1994.

[4] 保木本 晃弘, 中島 達夫, 渡部 卓雄. 移動計算機環境に適した拡張可能なオペレーティングシステムの構成技術, 第 11 回ソフトウエア科学会全国大会, 1994.

[5] 栗原 邦彰, 中島 達夫. サービスプロキシを用いた移動計算機環境の構築法, 第 11 回ソフトウエア科学会全国大会, 1994.

[6] Pattie Maes. Computational Reflection (Ph. D. Thesis), technical Report 87-2, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.

[7] Chris Maeda, Brian N. Bershad. Service wishout Servers, Carnegie Mellon University, Pittsburgh, PA 15213, 1993.

[8] 宮田重明, 芳賀敏彦共著. Tcl/tk プログラミング入門, オーム社, 1995.

[9] 中島 達夫, 保木本 晃弘. 移動計算機環境におけるアプリケーションとオペレーティング ・システム支援, 情報処理, 1994.

[10] 中島 達夫, 渡部 卓雄, 保木本 晃弘. 移動計算機環境に適した柔軟なソフトウエアの構築法, 第 49 回情報処理学会全国大会, 1994.

[11] John K. Ousterout. Tcl and the Tk Toolkit, Addison-wealey publishing company, 1994.

[12] Guido van Rossum. Python Reference Manual, Dept.CST,CWI,P.O.Box 94079 1090 GB Amsterdam, The Netherlands, 1994.

[13] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System, In Proceedings of the USENIX Mach Workshop October, 1990.

[14] 寺岡文男. VIP:ホスト移動透過性を提供するプロトコル, 日本ソフトウエア科学会コンピュータソフトウエア, Vol.10, No.4, 1993 年 7 月.

[15] 渡部 卓雄. リフレクション, コンピュータソフトウエア, Vol. 11, No. 3, 日本ソフトウエア科学会, 1994.

[16] Cache Kernel. http://www-dsg.stanford.edu/papers/cachekernel/

[17] Flux. http://www.cs.utah.edu/projects/flexmach/

[18] SPIN. http://www.cs.washington.edu/research/projects/spin/