メタレベルアーキテクチャによる移動計算機環境のための言語

A Language with Meta-level Architecture for Mobile Computing Environment

天野 憲樹
Noriki AMANO
n-amano@jaist.ac.jp

渡部 卓雄
Takuo WATANABE
takuo@jaist.ac.jp

北陸先端科学技術大学院大学 情報科学研究科*
Department of Information Science
Japan Advanced Institute of Science and Technology

概要

本研究で提案する言語LEADはメタレベルアーキテクチャによりアプリケーション本来の記述とシステムの動的変化に適応するための制御記述をモジュール化する言語であり、LEADにより移動計算機環境におけるシステム構成・環境の動的変化に適応可能なアプリケーションが容易に作成可能となる。本稿では移動計算機環境に対する従来の研究の問題点をふまえ、LEADの概要とその有効性について述べる。

はじめに

小型計算機の高性能化、PCMCIAや無線通信等の技術進歩、そして急速なネットワーク環境の整備と普及により、移動計算機環境 [3] が実現されつつある。移動計算機環境では計算機の移動にともなうIPアドレスの変化、PCカードの活線着脱等、計算機システムの構成やそれを取り巻く環境が動的に変化する可能性がある。こうした移動計算機環境に対し、これまではOSレベルで変化を吸収することにより上位のアプリケーションに対するシステムの透過性を実現してきた。このようなシステムの動的変化に適応可能なOSとしてSPIN [2]、Scout[6] 等が提案されている。しかしシステムの動的変化に対して、OSによる対応だけでは十分とはいえない。たとえば無線通信により遠隔地からメールの読み書きを行うことを考えてみる。この場合、OSレベルで計算機の移動にともなうIPアドレスの変化を吸収することは可能であるが、無線通信における通信の信頼性等を高めることはできない。もちろんOSによる動的なIPアドレスの付け替え等の処理は必要であるが、これだけでは十分とはいえない。このような場合、メールを一時的にスプールに貯めておき、有線によるネットワーク接続に切替えられた時にメールの読み書きを行う等の処理が考えられる。あるいはバンド幅の狭い無線ネットワークではメールの全文を送らず、サブジェクトのみ送信するなどといったことも考えられる。こうしたシステムの状況に応じたアプリケーションの振舞いは、OSによって実現されるよりはむしろアプリケーション自身の責任において実現されるべきである。なぜならば、システムの状況に応じて振舞いを変化するのはアプリケーションであり、変化の是非や度合はアプリケーションの性質やその処理の性質に依存するからである。しかし、このような処理を実現する機構を各アプリケーションがそれぞれ内部に実装することは現実的ではない。実装にかかる負荷が大きいだけでなく、アプリケーション本来の記述とシステムの状況変化に適応するための制御記述が混在することになり、プログラムの可読性や保守性を低下させる。これは既存の言語がシステム構成や環境に依存する部分としない部分と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。一方、システムの状況変化に適応する機構をライブラリ [4] として実現することも考えられる。しかしこうしたライブラリも個々のアプリケーションの細かい振舞いの記述に対応するには限界がある。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で記述できることが望ましい。このように移動計算機環境におけるシステムの状況変化に対応するにはOSとアプリケーションの双方がシステムの状況に対して適応可能である必要がある。しかしこれまでは前者の適応可能なOSのみに焦点があてられることが多かった。これに対し、本研究は後者のアプリケーションのシステム状況に応じた振舞いの実現に焦点をあてる。

以上の点を考慮し、本研究ではメタレベルアーキテクチャ [5, 7] によりアプリケーション本来の記述と適応動作のための記述をモジュール化する言語LEAD[1] を提案する。メタレベルアーキテクチャとは本来目的とする計算を行うベースレベルとそれを制御するメタレベルの二層構造からなる計算システムの機構である。このようなメタレベルアーキテクチャを用いることで各レベルは独立したモジュールとして分離され、プログラムの可読性、保守性、拡張性さらに再利用性を高めることができる。またLEAD

*923-12 石川県能美郡辰口町旭台 1-1

はアプリケーションのモジュール化を促進するだけでなく、システムの状況変化に適応する制御機構をアプリケーションに提供する言語システムでもある。この LEAD を用いることでシステム構成や環境の動的変化に適応可能なアプリケーションが容易に作成可能となる。LEAD による適応動作の機構とモジュール化により、各アプリケーションはその内部に適応動作のための制御機構を持つ必要がなくなり、そしてシステムの状況に応じた振舞いも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可能になる。LEAD はそれ自身で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あるが、このように他の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も可能である。これにより既存のアプリケーションに多様な環境に適応する機能を容易に付加することができるだけでなく、アプリケーションを新たにスクラッチから作成する必要がなくなる。

1 プログラミング言語 LEAD

LEAD はアプリケーション本来の記述と適応動作のための記述を分離する言語であり、システムの状況変化に適応するための制御機構をアプリケーションに提供する言語システム (以下、LEAD インタプリタ) である。LEAD インタプリタとアプリケーションはメタレベル、ベースレベルの二層構造で表現され、メタレベルアーキテクチャとして構成される。つまりベースレベルはアプリケーションであり、メタレベルは LEAD インタプリタである。LEAD インタプリタはアプリケーションにメタレベルとしてそれ自身の一部分を提供する。そしてシステムの状況に応じたアプリケーションの振舞いをこのメタレベルに記述し、LEAD インタプリタがそれらの振舞いを制御することで多様なシステムの状況変化に対応可能となる。さらにメタレベルアーキテクチャによりアプリケーションプログラムはシステム構成や環境に依存する部分としない部分の二つに分離されて高いモジュール性を達成し、可読性、保守性、拡張性および再利用性を高めることができる。

また LEAD は自己反映的な言語でもある (1.4 節参照)。つまり LEAD インタプリタはアプリケーションからの要求に応じて、それ自身の振舞いを改変し、それがアプリケーションの振舞いに反映する。このように LEAD インタプリタとアプリケーションはサーバ/クライアント、ベース/メタ、そして互いの振舞いに影響を与えあう自己反映的な関係である。

LEAD は言語として以下の三つの側面を有する。

1. LEAD_f :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側面。これらの振舞いは総称関数 (1.1 節参照) として LEAD インタプリタに登録される。総称関数は同一名称を持ち、かつ異なる振舞いのメソッド (1.2 節参照) を複数持つ。各アプリケーションは LEAD インタプリタへのインタフェースをとおして LEAD インタプリタに対し、総称関数の呼び出しを要求する。アプリケーションからの要求を受け取った LEAD インタプリタはシステム構成・環境に応じたメソッドを選択し、アプリケーションにそのメソッドを送る。そしてメソッドを受け取ったアプリケーションがメソッドを実行する。
2. LEAD_t :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側面。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ことができる (1.5 節参照)。
3. LEAD_d :LEAD インタプリタ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機構を記述する側面。このディスパッチ機構も LEAD によって記述される。メソッドのディスパッチ機構はいわば LEAD のメタオブジェクトプロトコル [7] としてユーザに公開されているものであり、LEAD を用いてディスパッチ機構を容易にカスタマイズすることができる。またメソッドのディスパッチ機構は後述のポリシに基づいて実現され、アプリケーションからの要求により動的に変更することが可能である (1.3 節参照)。

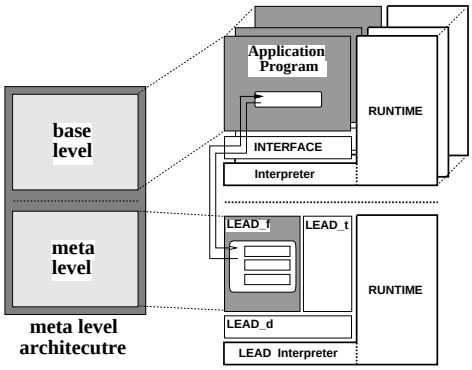

図 1: LEAD のメタレベルアーキテクチャ

さらに LEAD はプログラムのリモート実行機能を提供している。この機能によりエージェント指向的なスクリプトを作成し、リモートホスト上で実行することができる。LEAD の提供するリモート実行はシステムの状況に応じて自動的にリモート実行を行う、リモートホスト上にないプログラムの実行も可能といった利点を有する。

1.1 総称関数

総称関数はシステムの状況に応じた処理の総称であり、異なる振舞いを持つ複数のメソッドへのインタフェースである。ある処理について個々のメソッドを記述する前に、その処理を総称関数として定義しなければならない。総称関数の定義では以下の指定を行う。

1. 総称関数名
2. アプリケーション記述言語名
3. 状態要素のリスト

状態要素とはその名前に関連づけられたシステム資源の情報をモニタリングする属性であり、例えば `memory`, `battery` 等がある。状態要素はあらかじめ LEAD 中に定義されているものもあるが、ユーザが新たに定義することも、再定義も可能である。ここで指定した状態要素の状態に基づいてメソッドが選択される。状態要素は複数指定することができ、その優先度が高い順に左から指定する。この総称関数の定義により、LEAD インタプリタ中にこの総称関数に対する状態要素の優先度を示す優先度数リストが生成さ

れる。優先度数は指定された状態要素の数を n とすると、2^n−状態要素の指定順で算出される。

1.2 メソッド

メソッドは個々の状況に応じた処理の実体であり、アプリケーションプログラムの一部を含む。通常一つの総称関数について複数の異なる振舞いのメソッドが存在する。メソッドの定義では以下の指定を行う。

1. メソッド名
2. アプリケーション記述言語名
3. ディスパッチ条件リスト
4. 実行ロケーション
5. 実行コード

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、総称関数の定義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。ここで使用可能な状態要素はこのメソッドに対する総称関数定義で指定した状態要素に限られる。また複数の条件を記述することも可能である。次に実行ロケーションとは続く実行コードを実際に実行するホストを指定する。また実行コードとはアプリケーション側に送られ実行されるプログラムコードであり、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。このメソッド定義により生成されるデータ (メソッドリスト) は LEAD インタプリタ中のメソッドを登録するテーブル (メソッドテーブル) 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。

このようなメソッドおよび総称関数を LEAD インタプリタに登録する言語としての側面が LEAD_f である。

1.3 メソッドのディスパッチ

LEAD におけるメソッドディスパッチのメカニズムは総称関数定義によって生成される状態要素の優先度数リストと、メソッド定義によって生成されるメソッドリストが追加されたメソッドテーブルを参照・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機構は LEAD によって記述され、これは 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あると、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メソッドテーブル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・環境に適したメソッドが選択される。LEAD はメソッドを選択する際のポリシとして標準で三つのポリシを提供している。それらのポリシは以下の二つに大別される。

1. 選択メソッドポリシ
2. 必須メソッドポリシ

メソッドには、ある特定のシステム構成・環境の場合にだけ必要なものと、いかなる構成・環境の場合にも必要なものの二つが存在する。選択メソッドポリシはディスパッチ条件を満たすメソッドを選択するが、条件を満たすメソッドがなかった場合にはメソッドを選択しない。この選択メソッドポリシには and ポリシと or ポリシがある。これらは直観的に明らかなようにディスパッチ条件リストの全ての条件が真 (and) か、または少なくとも一つ以上が真 (or) の場合、メソッドが選択される。つまりメソッドは選択されるか否かの二者択一である。反対に必須メソッドポリシはどのような条件下でも必ずメソッドを選択する。必須メソッドポリシは以下のルールに基づいて実現されている。

(a) 各状態要素に対する条件が偽となるものの優先度数の合計値が最小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優先度数の合計値が最大であるものを優先する。

つまりルール (a) による合計値が同じであるメソッドについて、ルール (b) を適用することでメソッドの特定化を行う。これによりメソッドが一意に決定できる。LEAD はデフォルトのディスパッチポリシとして必須メソッドポリシをとる。and や or といった単純なディスパッチポリシではメソッドが一意に決定できない、あるいはメソッドが選択されない場合があるからである。

1.4 自己反映的な動作

LEAD インタプリタとアプリケーションは自己反映的な関係にあり、互いの振舞いの変化が互いの振舞いに影響を与えあう。自己反映的 [5, 7] とは自己の構成や計算過程を内部に持ち、その表現の参照・変更が可能であり、かつその変更が自己の構成・計算過程に反映するようなシステムを指す。前節で述べた LEAD のディスパッチ機構は LEAD 自身で記述されており、そのポリシは総称関数として定義されている。そしてインタフェースをとおしてアプリケーションから使用するポリシを LEAD インタプリタに要求することができる。ポリシの要求を受けた LEAD インタプリタは該当するポリシを実現するメソッドをメソッドテーブルから選択し、LEAD インタプリタ上で実行する。これによりポリシは新たに選択されたポリシに変更され、これ以降は新しいポリシに基づいてディスパッチが実行される。つまりポリシの変更はアプリケーションのメソッド選択に反映する。このようなポリシの変更を要求する処理を総称関数として定義すると、LEAD のディスパッチポリシとそれにより選択されるアプリケーションのメソッドが交互に改変を繰り返すメタ・サーキュラな構造を作ることが可能になる。すなわちメソッドとポリシは完全に独立しており、任意の場所でポリシの変更ができるとともに、ポリシが総称関数として定義されていることから、ユーザは他の総称関数と同様にポリシを自由に変更・追加することが可能となる。

1.5 適応可能なスクリプトの実現

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 と名づけた LEAD の一つの側面を指す。この LEAD_t により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スクリプトを容易に作成できる。つまりシステム構成・環境に応じてアプリケーションを組替えたりする処理が記述可能である。しかも LEAD_t によるスクリプトは通常のスクリプトよりも一段深いレベルの状況に応じ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、各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組替えと同時に各アプリケーション中の振舞いも改変する。これはシステム状況に応じたスクリプトを総称関数として定義することで容易に実現される。

2 例題 :テキスト ・エディタ

本研究では例題として LEAD と Tcl/Tk を用いて、システム状況に応じて振舞いを変えるテキスト ・エディタを作成した。このエディタはシステム構成や環境に応じて以下のように振舞いを変える。

ディスプレイのサイズに応じた振舞いとしてアプリケーションのメインウインドウ、ダイアログウインドウ、文字のサイズを動的に変更する。さらにウィジェット (ウインドウを構成するボタン等の部品) の配置も変更する。

次にメモリ容量とバッテリ残量に応じた振舞いとして文字入出力バッファの伸縮を行う。入出力バッファのサイズを大きくとれば、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ともな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。ここではメモリの節減を優先している。

そしてバッテリ残量に応じた振舞いとしてファイルの自動保存回数の増減を行う。このエディタはファイルの自動保存機能を持ち、バッテリの残量が少ない時には自動保存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。

このようなシステム構成 ・環境に応じ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。それは LEAD インタプリタがシステムの状況変化に対する適応動作のための制御機構を提供しているため、アプリケーションではそのための制御記述を一切書く必要がないからである。LEAD によって記述されるのは状況に応じた振舞いを実現する総称関数とメソッドだけであり、エディタ全体からみた LEAD の記述量はごくわずかである (表 1 参照)。しかも LEAD の記述量のうちの大半は Tcl/Tk を使用して書くことができる。これはメソッドの実行コード部分が最も記述量が多いためである。

Tcl/Tk の行数	1196
LEAD の行数	85
(メソッドの実行コードの行数)	(65)

表 1: エディタにおける Tcl/Tk と LEAD の記述量

さらに上記のシステムの状況に応じた振舞いはすべて独立したモジュールとして定義されており、アプリケーションから分離され、各モジュール単位での再利用が可能である。例えばディスプレイのサイズに応じた振舞いを実現するモジュールは、このエディタ以外の Tcl/Tk のアプリケーションでも容易に再利用可能である。これらのモジュールはアプリケーション本体とリンクする必要はない。つまり状況に応じた振舞いを実現するモジュール群はアプリケーション本体とは独立して単独で作成することができ、アプリケーションにインクリメンタルに付加することができる。

このように既存のアプリケーションに対して LEAD を用いることで、少ない記述量でシステム構成 ・環境の変化に適応可能なアプリケーションの振舞いを付加することが可能になるうえ、それらの振舞いは独立性の高い再利用可能なモジュールとして作成することが可能になる。

3 実装

LEAD のプロトタイプは Scheme に似た文法を持つインタプリタ言語として SunOS および BSD/OS 上に実装した。このプロトタイプは他のアプリケーションとやりとりするためにソケットによるプロセス間通信を採用している。そのためアプリケーション側にも LEAD インタプリタとの通信インタフェースが必要になる。本研究では Emacs Lisp、Perl、Tcl/Tk、Python で作成されたアプリケーションに対する LEAD インタプリタとの通信インタフェースを実装した。また例題としてあげたテキスト ・エディタも同一プラットフォーム上で実装した。

4 まとめ

移動計算機環境におけるシステム構成 ・環境の変化に適応可能なアプリケーションを作成 ・支援するために有効な言語 LEAD を提案した。例題の作成をとおして LEAD により少ない記述量で既存のアプリケーションにシステムの状況変化に応じた振舞いを付加することができること、プログラムのモジュール性、可読性、保守性、拡張性そして再利用性が高まることを示した。

今後の課題としては、LEAD インタプリタとアプリケーション間の通信インタフェースの改善、プログラムのリモート実行に有効なプロトコルの考案と実現、そしてより効果的なメソッドのディスパッチポリシの考案と実装があげられる。

参考文献

[1] 天野 憲樹, 渡部 卓雄. 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, Research Report, IS-RR-96-0014S, School of Information Science, Japan Advanced Institute of Science and Technology, May, 1996. http://www.jaist.ac.jp/ ~n-amano/paper.html

[2] Bershad, B., Savage, S., Pardyak, P., Sirer, E.G., Becker, D., Fiuczynski, M., Chambers, C., Eggers, S. Extensibility, Safety and Performance in the SPIN Operating System, In Proceedings of the 15th ACM SOSP, pp. 267–284.

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

[4] Maeda, C., Bershad, B.N. Service wishout Servers, Carnegie Mellon University, Pittsburgh, PA 15213, 1993.

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

[6] Montz, A.B., Mosberger, D., O'Malley, S.W., Peterson, L.L., Proebsting, T.A., Hartman, J.H. Scout: A Communications-Oriented Operating System, Department of Computer Science The University of Arizona Tucson, AZ 85721.

[7] 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.