美しい日本の ML コンパイラ
―― 最強のプログラミング言語」を理解する――

1. 背景

コンピュータは動かない。特にソフトウェアはバグが多い。しかし、ソフトウェアのバグを防止する方法としては、(1) プログラマができるだけ注意する、(2) 人海戦術でテストする、(3) 問題が発見されたら修正する、(4) 発見される問題が一定以下になったらリリースする、(5) リリースしてから問題が発見されたらパッチを配布する、というまったくad hoc なアプローチが現在の主流である。このようなソフトウェア工学的なアプローチは、既存のプログラミング言語や開発環境にあまり変更の必要がないという利点はあるが、抜本的解決にはならない。

より drastic なアプローチとして、C や C++のような``legacy"あるいは``patchy"な（々はオペレーティングシステムを記述するための）低水準言語ではなく、より適切に設計された高水準言語（たとえばMLやLISP）を適材適所で使用する、という発想は自然である。しかしながら、社会的慣性や技術者養成の問題から、これらの言語が C や C++と比較してまったくの minority であることは否定のしようがない。JavaやC#における自動メモリ管理やgenericsのように、ML や LISP のような言語のアイディアを C/C++風言語に「技術移転」するアプローチも活発であるが、やはり継ぎ接ぎの感が否めず、GC が保守的になったり、generics に不自然な制限が要求されたりといった問題が存在する。

2. 目的

開発代表者らは、より地道ではあるが自然な「正面突破」のアプローチとして、お堅いプログラミング言語と思われている ML を少しでも広めるために可能な活動をしている。そのような活動の一環として未踏ソフトウェア創造事業では、学生やプログラマでも詳細まで理解できる簡潔な ML コンパイラを開発した。

開発の目的は、「ML は何をやっているのかよくわからないから使いたくない」という疑念を払拭することにある。開発代表者の経験では、そのような考え方はかなり根強く、ML が広まるもっとも大きな妨げの一つとなっている。たとえば、「ML は高階関数があるからインタプリタでしか実装できない」という迷信はよく流布している。これは関数クロージャの概念を知らないためであるが、それを教えると、今度は「ML の関数はクロージャが必要だから効率がよくない」といった誤解が生まれる。しかし、実際には自由変数のない既知関数は「普通に」呼び出すことができるので、たとえば C 言語と比較しても（そもそも C 言語で記述できる関数については）同一の処理を行っており、効率が悪いということはない。このような迷信や誤解を十分に払拭するためには、研究者や専門家でなくとも（多少の ML プログラミングの経験さえあれば）詳細まで理解できる ML コンパイラを提供・解説することが必要である。

3. 開発の内容

プログラミング言語 ML のサブセットについて、以下の特徴をもつコンパイラを

開発した。

(1) 実装が簡潔である。実装言語としては Objective Caml を使用した。ソースコードの行数としては、各々のモジュールが 10 行～300 行程度、全体で 2000 行程度となった。実装が**簡潔**だからといって**容易**であるわけではないことに注意されたい。むしろ、必要かつ十分な機能を適切に実装するために、高度な背景知識にもとづく取捨選択や試行錯誤が必要となった。
(2) 性能のよいコードを生成する。単純なアプリケーションについては等価な C プログラムと同等以上、複雑なアプリケーションについても Objective Caml と同等以上の性能を達成した。ターゲットプラットフォームとしては、多数の教育機関で利用可能な RISC CPU である SPARC プロセッサを採用した。
(3) 自明でないアプリケーションを実行できる。結果がわかりやすくおもしろいというメリットから、レイトレーシングを使用した。

コンパイラの記述言語としては、やはり ML の一種である OCaml を使用した。

どのようなサブセットになるかは進捗状況にもよると予想されたが、提案プロジェクトの期間では、以下の機能を実現した。算術演算、組、破壊的代入の可能な配列、高階関数、再帰と末尾呼び出し、型推論。以上の機能があれば、単純なアプリケーションやレイトレーシングを実行するには十分である。また、ML や LISP のようないわゆる関数型言語のコンパイラの基礎を解説でき、より高度な機能の実装も理解が容易になる。将来の方向としては、ゴミ収集、多相型、例外処理、再帰データ型とパターンマッチング、モジュールシステムなどの実装を検討している。

提案プロジェクトで開発した各々のモジュールは、コンパイラの個々のフェーズに対応する。コンパイラの大体の構成は図 2 の通りである。全体の方針は、ML という高水準言語とアセンブリという低水準言語との大きなギャップを一つずつ取り除いていくことにある。適切な中間言語を設定すれば、これは単純な記号変換の連続となる。

(1) まず、ソースコードを字句解析・構文解析する。これらの解析には ocamllex と ocamlyacc を使用したが、将来は packrat parsing という、より洗練された手法も実験してみる予定である。
(2) 次に参照を利用した単一化による型推論を行う。
(3) 計算の中間値をすべて明示化する K 正規化という変換をする。これは、ML では任意のネストした式が書けるのに対し、アセンブリでは一度に一個の計算しかできない、というギャップを取り除くためである。
(4) 変数や関数の名前が衝突しないようにつけかえる（α 変換）。
(5) インライン展開、定数畳み込み、不要定義除去を繰り返す。
(6) アセンブリには（当然ながら）ネストした関数がないので、ML のネストした関数を平らにするクロージャ変換を行う。このクロージャ変換では、自由変数のない既知関数の呼び出しをそれ以外の関数呼び出しと区別し、不要になったクロージャは生成しない。
(7) クロージャ生成や組・配列の読み書きを load/store に変換し、レジスタが無限にある仮想マシンのコードを生成する。
(8) 最後に、一定の呼び出し規約にしたがいレジスタ割り当てをして、SPARC

アセンブリを生成する。ただし、生成するコードの内部では SPARC プロセッサのレジスタウィンドウは使用しない（外部関数を呼び出すときには使用する）。

このコンパイラにおいて動作する最大公約数、フィボナッチ、アッカーマン関数、ベクトルの内積、行列積、マンデルブロー集合、ハフマン符号といった単純なプログラムと、自明でないアプリケーションとしてレイトレーシングを提供した。ただしレイトレーシングはオリジナルではなく、既存のプログラムを利用した。さらに、これらのプログラムを自動でコンパイル・実行して結果をチェックする Makefile を提供し、いわゆる extreme programming でいう自動テストを実現した。

開発内容の詳細については、Web サイト（http://min-caml.sf.net/）を参照されたい。

4. 従来の技術（または機能）との相違

我々の対象言語は単純な ML サブセットだが、実装が数千行以下、性能は GCC や OCaml に匹敵、かつ自明でないアプリケーションを実行できるというコンパイラは、（開発者らの知る限り）他言語でも今までに無い。

5. 期待される効果

開発したコンパイラおよびその解説を高等教育機関で利用および一般公開することにより、ML ないしプログラミング言語一般について高度な理解をもった人材を育成、究極的には高信頼・高品質ソフトウェアの開発につながることを期待している。

6. 普及（または活用）の見通し

開発されたコンパイラは、すでに東京大学理学部情報科学科など複数の高等教育機関の授業や演習で使用されている。また、オンライン・チュートリアル「速攻 MinCaml コンパイラ概説」および ML 自体の紹介「超特急: 一時間でわかる ML 超入門」を公開（http://min-caml.sf.net/）、関連するメーリングリストでアナウンスし肯定的反響をえた。

7. 開発者名（所属）

住井英二郎（ペンシルバニア大学）
住井真紀子（フリー）

（参考）開発者URL

http://www.cis.upenn.edu/~sumii/