4A-05

Minisatとの比較によるHaskell SATソルバーの高速化

楢崎 修二[1]
長崎大学

1. はじめに

近年高速化が著しい SAT ソルバは種々の制約問題を解くための重要なツールである。そのため多くの言語で利用できることが望ましい。現在主に利用されるものは C++言語による Minisat とその派生プログラムである。Minisat はソースが公開され、解説論文 [2] があるため、いくつかの言語に移植されている。

手続き型・オブジェクト指向型言語とは大きく違うパラダイムに基づく純関数型言語 Haskell 言語が近年注目を浴びている。型レベル計算、関数合成による言語機能の拡張、モナドによる副作用計算の閉じ込めなど、他の言語には見られない機能を多く有し、その一方で種々のライブラリや高度な最適化を行うコンパイラの存在などにより人工知能分野を含めた様々な分野への応用・実用も見られるようになってきた。更なる分野への適用のため SAT ソルバーの導入が望まれる。FFI を介して他言語ライブラリを利用することもできるが、導入コスト、問題毎のヒューリスティックの導入、内部データ変換コストなどを考えると Haskell 言語自身で必要な機能を実装することが理想である。

先に述べたように SAT ソルバーは Minisat のソースが公開されているため、十分高速なものを実装することが容易であると考えられるにも関わらず、これまでの Haskell による SAT ソルバーは十分な性能を達成できていない。特に単にコンパイラの最適化の程度の違いでは説明できない計算量の差が存在していた。そこで本研究では Minisat の忠実な移植を行うことで Haskell による高速 SAT ソルバーの実現について検討する。

2. Minisat の基本構造

代表的な高速 SAT ソルバーである Minisat 1.14 は以下の特徴を持つ [2]。

- 監視リテラル対による制約伝播の高速化
- VSIDS による変数活性度計算
- 矛盾理由解析時の利用回数に基づく節活性度の計算とそれを基にした学習節の半数を削減する学習節管理
- 幾何級数（現在は Luby 列）によるリスタート間隔制御

これらを実現するため、主として以下のデータ構造を用いる。

- ソルバ クラス
- ブール制約 クラス
- 可変長ベクタ

最初の２つは対象とする制約を拡張可能にするための Minisat のソフトウェア構造を定義するものである。最後のものが変数や節や活性度などを表現するための基本となるデータ構造である。これは以下の仕様を満足しなければならない。

- $O(1)$ の手間での添字指定要素の参照・更新
- $O(1)$ の手間での要素の追加
- （リテラルや割当を表す） 整数、（変数を表す）正整数、節、（節リストを表す）可変長ベクタを要素に取る多相性
- （監視リテラルリスト実現、学習節管理のための）小さな手間での指定要素の削除
- 単位伝播リテラルのためのキューおよび割当履歴のためのスタックとしてのインターフェース

これらの仕様を満足するデータ構造を選択できれば、他言語への移植は難しいことではない。

3. Haskell 言語と SAT ソルバー

Haskell 言語の特徴の中で本研究に関連するものについて紹介する。まず遅延評価および GC を持つ言語である。そのためポインタは直接値を指すとは限らず、ヒープ上にポインタオブジェクトを割り当てる必要がある。データ型の中にはいわゆる即値に対応するものがありこれを “unboxed” という。コンパイラは関数呼出しにおける中間生成データの削除、インライン展開などの最適化を行う。簡単なプログラムで（データの内部表現に踏み込んだ操作を行う）コンパイラ固有の機能を使った場合には gcc に匹敵するという報告がなされている。

多くの関数型言語で用いられるリストは要素参照の手間が高い。また、破壊的操作の度にヒープメモリを消費するという問題もある。そのため高速なプログラムを作るには、$O(1) \sim O(\log n)$ で破壊的操作を可能にする適切なデータ構造を選ぶ必要がある。

次に Haskell 言語上に作られた SAT ソルバーについて説明する。

funsat は Haskell 言語によるもっとも早期に作られたプログラムである [5]。文献 [2] を参考に作られたものであり、関数名などに対応が見られる。funsat では集合計算用の immutable なデータ構造を採用している。これによって基本操作の計算量はある程度は減らすことができるが、代入の度にヒープメモリを使用することになり、キャッシュ率の低下、GC 時間の増大などの問題が避けられない [3]。これらの問題から、図 1 に示すように、Minisat を基に作成されたにも関わらず性能は遥かに及ばない。

sih4 は我々が昨年まで開発していたプログラムである [3]。funsat と同じく、文献 [2] で用いられた多くのアルゴリズムを実現したものである。funsat で用いられていない、モナド内で破壊的代入可能な配列 `Vector` 型を用いることで、基本操作の高速化とヒープ使用量の減少を目指した。Vector 型は正格評価済みのデータに対象を限定した Unboxed とそうでないものの 2 種類が提供される。`Vector.Unboxed` は遅延評価が出来ない代

Improvement of a SAT solver in Haskell by comparing with Minisat

[1] Shuji Narazaki
Nagasaki Univeristy

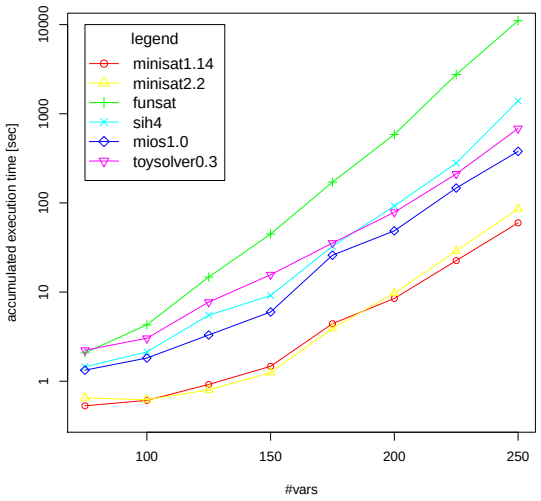

図 1 3-SAT 問題の変数数に対する実行時間の変化

わりにポインタではなく他の言語と同様に値そのものを配列内に格納するため、破壊的操作時にヒープメモリの動的な割当は起こらない。これによってヒープ使用状況を大きく改善することができる。しかし十分な性能を得ることができなかった [3]。

toysolver は近年開発が進められている、SAT を含む汎用の制約問題解決器である [7]。十分な検討ができていないが内部構造は Minisat を基にしたものではない。

4. 実装方針

今回我々は sih4 で採用した `Vector.Unboxed` を用いるという方針を基にして、Minisat1.14 を忠実に移植することにした。特徴としては以下が挙げられる。

- sih4 では節に番号を振り ID として使用することで節の同値性を判定していたが、ポインタレベルでの同値性判定を行うコンパイラの拡張機能 `reallyUnsafePtrEquality#`を用いて、$O(1)$ での節比較と ID 格納領域の削減を図った。
- 節削減に関わる計算量的な差はないと考えられるので、sih4 では十分に検討していなかった学習節管理として Minisat のものを忠実に移植した。
- 監視リテラルを実現するデータ構造を節を「ポインタ」でつなぐものにし、節以外のデータ構造の使用を減らした。
- キューおよびスタックはそれぞれ独立したデータ型とした。また SAT ソルバーにおいてはそれらの格納数および格納される値がどちらも変数数で定まる上限を持つことから、固定長の `Vector.Unboxed` によって実現することでヒープ使用量を削減した。

以上により、Minisat に忠実なアルゴリズムおよび（できる限り）同等の操作計算量を持つデータ構造が選択できたと考えられる。なお、逸脱点として、多相性のもたらす速度低下を避けるためにソルバクラス・ブール制約クラスによる拡張性を省略して SAT 問題に特化させた。

5. 評価

SAT ソルバーのベンチマークとして使われた、変数数の違う 3-SAT 問題 [6] を各変数数ごとに 100 問解いた合計時間により、

表 1 ヒープ使用状況の比較（単位：MB）

ソルバー	ヒープ割当量	最大常駐量	全メモリ
Minisat	-	-	20
sih4	332,743	9.8	29
mios	1,943	1.4	5

作成プログラム `mios`[4] の評価を行った。実行速度に関する結果を図 1 に示す。また、ヒープ利用プロファイルを表 1 に示す。

- 変数数に対する計算量の増加状況は Minisat と同等と見られる。これは funsat, sih4 に比べて改善でき、目標が達成できた。
- toysolver もほぼ計算量の増大率は mios と同じである。ただし、実行速度は mios の方が数割速く、Haskell 言語で書かれた SAT ソルバとしては最速である。
- Minisat に対する速度比は 1/3〜1/4 倍である。これは言語の差を考慮すると妥当な範囲であると思われる。
- 以前のソルバでは実行の経過に連れヒープ使用量（最大量）は単調に増していたが、mios ではどちらも低く抑えられた。

6. まとめ

今回、忠実な Minisat の移植をすることにより、これまでで最も高速な Haskell 言語による SAT ソルバを開発することができた。計算量的にはほぼ Minisat に匹敵すると考えられる。速度差に関しては C 言語で記述された関数の呼び出しなどの Haskell 言語の枠からはみ出た拡張機能の導入などを行わなければ、これ以上の改善は難しいと考えている。しかし、LBD や community に基づく変数・節活性度、phase saving、local restart といった種々の研究成果を導入することで Minisat に対し 2 倍程度の速度低下に抑えることは十分可能であると思われる。これは SAT competition などで競うには全く不十分ではあるが、Haskell 言語のためのツールとしては有用であると考える。

今後の課題としては、より大きな問題での評価や、上に挙げた Minisat1.14 以降の研究成果の導入がある。

参考文献

[1] Haskell Language, home page, `https://www.haskell.org/`
[2] N. Een and N. Sorensson, “An extensible SAT-solver,” in *6th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT2003)*, pp. 502-518, 2003.
[3] 楢崎修二、GC の削減を目標とする関数型言語 Haskell による高速 SAT ソルバの実装、IPSJ2014, 情報処理学会、2B-6, 2014-03.
[4] Shuji Narazaki, mios, `https://github.com/shnarazk/mios`
[5] Denis Bueno, funsat, `https://hackage.haskell.org/package/funsat`
[6] SATLIB, benchmark problems, `http://www.cs.ubc.ca/〜hoos/SATLIB/benchm.html`
[7] Masaki Sakai, toysolver, `https://github.com/msakai/toysolver`