1F3-2

関数型言語Haskellによる SAT ソルバーの実装と比較

Implementation and Comparison of an SAT Solver in Functional Language Haskell

楢崎修二 *1
Shuji Narazaki

*1長崎大学
Nagasaki University

We are developing a new SAT solver in functional language Haskell. Though Haskell has many advanced features like test framework, type-level computation and meta-level computation, the previous SAT solvers written in Haskell had very poor performance, even if they used same algorithms with solvers in C/C++. To solve the problem, we tried to make a complete translated version of MiniSat 1.14 which is written in C++ and to compare them carefully. Though benchmark we could find and fix some problems in the previous solvers in Haskell and got the fastest SAT solver.

1. はじめに

関数型言語は安全性や拡張性を実現する強力な型システムを持ち，研究から実用まで幅広い注目を浴びている．その代表的な言語として Haskell[Haskell] が挙げられる．より幅広い応用のための一つの機能として制約問題を解く SAT ソルバーの提供が望まれるが，Haskell で記述されたものには十分な性能を持つものがなかった．本研究では代表的な SAT ソルバーである MiniSat を忠実に実装することで，どの程度の性能が得られるのかを検討し，この分野での Haskell 言語の応用の可能性を検証する．

2. 背景

命題論理式の充足可能性の判定器を SAT ソルバーと言う．監視リテラルリストや VSIDS といった実装手法や学習節の評価手法などの研究の進展により目覚ましい速度向上がなされたため，様々な充足問題への応用が進みつつある [鍋島 10, Biere09]. MiniSat はそのさきがけとなった代表的な高速 SAT ソルバーである．ソースが公開されていることに加え，実装に関する論文が発表されていることから，その後のソルバーや他言語への移植の開発ベースとして用いられることが多い．

優れた型システム，遅延評価，型レベル計算などの特徴的な機能を持ち，安全なソフトウェアの開発を可能にするものとして注目を浴びている関数型言語の代表的な言語である Haskell においても MiniSat を基にして作成されたソルバーが公開されているが，計算速度に大きな問題があった．例えば 2008 年に公開された funsat[Bueno] は MiniSat で数秒で解ける問題に対して 1000 倍以上の速度差があり，コンパイラの性能比では説明できない計算量の違いが生じている．近年開発されているソルバー [Sakai], [楢崎 14] であっても 2003 年に公開された MiniSat-1.14 に匹敵するような性能は得られていない．

これは，手続き言語またはその拡張であるオブジェクト指向言語間であれば，制御構造，提供されるライブラリなどに多くの共通性が存在することからプログラムの移植作業は一般に容易であるのに対し，（純）関数型言語への移植では，制御構造やオブジェクトのメモリ中の表現，提供されるライブラリの性質などに性能を劣化させる要因が存在しているからである．例えば，破壊的操作を可能にする配列は多くの言語と違い，言語の提供するプリミティブではなく，コンパイラの提供する低レベルな機能やデータ構造から提供されるライブラリである．また入出力という基本機能であっても，遅延評価を基とする言語上で，メモリリークを起こさず効率よく実現すればよいかは現在も議論を起こす課題であり，この数年内でも新たなフレームワークが作成されているという状況である．このように，言語のプリミティブな機能・ライブラリ・その上に存在するアルゴリズムが階層構造として分離できる場合にはアルゴリズム層のみを対象にすることで成立するはずの移植という作業は，Haskell においてはその妥当性を十分に検討しながら行うことが必要になる．例えばリスト型の安易な使用は大きな計算量の差になって現れる．

3. 先行研究と研究目的

ここで，我々自身による先行研究について述べる．これまでの Haskell 言語による SAT ソルバーの開発を通して，我々は MiniSat のプログラムで使用されている大部分のデータ構造が整数配列によって実装されることを確認した．これに対応する，ヒープ上のオブジェクトへの参照を伴わなず，破壊的な代入操作が可能な配列型は Haskell 言語でも提供されているため，計算量の違いなしで移植できるはずである．そこで，論文 [Een03] で解説されている MiniSat のアルゴリズムの移植を行った．得られたソルバーはこれまでのものに比べて 10 倍以上の高速化ができたものの，規模の大きな問題では依然計算量に違いが生じていた [楢崎 16].

このような差が生じる理由として，MiniSat の高速性は論文では触れられていない細かな実装技術（例えば節内リテラル順序，簡略化における節の並べ替え方針，学習節の削除方法など）によるところが大きいと考えられる．実装レベルの差異は Haskell 言語によるソルバーでは無視されてきたものである．

そこで本研究ではプログラムコードレベルで可能な限り忠実な移植をすることにした．そのため，移植対象を論文から実際のプログラムである MiniSat-1.14 として選んだ．これは MiniSat1.14 が現在のソルバーの基礎となっていること，論文からの変更点が少ないこと，MiniSat 2.0 以降にはメモリ管理に関するコードが含まれているため単純には変換できないこと，による．なお，現在の開発状況はほぼ全てを移植し終わり，大きな違いとして残っているのは，

連絡先: 長崎大学工学研究科，narazaki@nagasaki-u.ac.jp

- 2 リテラル節に特化した節表現への対応
- SAT 問題以外の問題クラスへの対応を可能にするためのクラス構造による拡張性
- 問題ファイルの入力・解析方法，BCNF 形式やファイル圧縮形式への対応，実行中の統計データの表示などの入出力

である．逆に，単純な前処理（正（負）リテラルのみ出現する変数への求解前の割当），phase-saving（バックトラック後の決定による割当において前回の割当値を使用する）が Minisat-1.14 からの逸脱分として追加している．

以下では本研究で開発したソルバーを mios-1.1 と表記する (Minisat-based Implementation and Optimizaiton Study version 1.1)．先行研究 [楢崎 16] での実装を mios-1.0 として表記する．

4. 実験評価

まず簡単な問題によってヒューリスティクスに依存しない実装基盤の基本性能の評価を行う．次にヒューリスティクスの移植の妥当性を実際的な規模の大きな問題を対象として検討する．実験環境は CPU : Intel Core i7-3930 (3.8GHz), メモリ: 16GB, OS : Linux 4.4 (arch linux 64bit), コンパイラ : ghc 7.10.3 である．乱数による変数選択確率や活性度減衰度などのソルバーの持つパラメータは同じ値を用いた．

4.1 入出力および基本機能の評価

現在のソルバーの高速化の主たる要因はヒューリスティクスの改善によるものである．そのため，実装言語とヒューリスティクスとの両方ともに違うソルバー間での比較結果を評価するのは困難となっている．そこで，まずヒューリスティクスの影響を排除できる問題を考え，入力，制約伝搬や単位節の検出などのソルバーの基本操作の速度を比較する．そのために以下の 3 種類の問題を用意した．いずれも決定による選択（従ってバックトラック）を起こさない問題である．

- 単連鎖 (3) 問題 : $1 \wedge 2 \wedge (\bar{1} \vee \bar{2} \vee 3) \wedge (\bar{2} \vee \bar{3} \vee 4) \wedge \cdots$

  リテラル 1 および 2 の単位節と $2 < N \leq n$ に対して 3 つのリテラル $-N+2, -N+1, N$ を含む $n-2$ 個の節からなる問題である．連続する n 回の含意による割当の連鎖のみで充足する．求解中に監視リテラルリスト間での節の移動は起きず，割当履歴が単調に増加する．他の問題と比べてファイルサイズが小さく，全ての節を走査する時間，割当データ構造の参照・更新のための処理が主となる問題である．各節の長さが 3 なのは，MiniSat-1.14 での 2 リテラル節への特化による実装の違いの影響を排除するためである．

- 0 連鎖問題 : $1 \wedge (1 \vee 2) \wedge (1 \vee 2 \vee 3) \wedge (1 \vee 2 \vee 3 \vee 4) \wedge \cdots$

  リテラル数を n としたとき，リテラル 1 から n を全て含む長さ n の節が各 1 回出現する問題である．決定レベル 0 での 1 回の含意により全ての節が充足し，監視リテラルリスト内での節の移動を必要としないため，(前処理をしない場合は）節データの入力と構成に要する時間が主となる問題である．

- 三角連鎖問題 : $1 \wedge (\bar{1} \vee 2) \wedge (\bar{1} \vee \bar{2} \vee 3) \wedge (\bar{1} \vee \bar{2} \vee \bar{3} \vee 4) \wedge \cdots$

  リテラル 1 からなる単位節と，$1 < N \leq n$ に対してリテラル $-N+1, -N+2, \cdots, -1$ および N を含む $n-1$ 個の節からなる問題である．含意による割当の連鎖のみで充足する問題であり，高度な前処理が行われない場合には，監視リテラルリスト間での節の大量の移動が必要となる．

表 1: 単連鎖 (3) 問題の実行時間（単位：秒）

n	8K	20K	80K	200K	800K
バイト数	146K	398K	1.7M	4.5M	19M
MiniSat	0.010	0.017	0.060	0.120	0.310
入力部	0.007	0.013	0.043	0.087	0.243
求解部	0.003	0.004	0.017	0.033	0.067
mios	0.040	0.110	0.597	3.113	39.847
入力部	0.037	0.107	0.580	2.963	39.710
求解部	0.003	0.003	0.017	0.050	0.137
toysat	0.190	0.380	1.277	3.017	12.647

表 2: 0 連鎖問題の実行時間（単位：秒）

n	1000	2000	4000	6000	8000
バイト数	1.9M	8.0M	35M	81M	145M
MiniSat	0.057	0.160	0.424	0.848	1.449
消費メモリ	17M	23M	47M	85M	139M
mios	0.248	1.232	8.875	29.383	69.123
入力部	0.243	1.227	8.863	29.370	69.103
求解部	0.000	0.000	0.003	0.000	0.000
toysat	2.34	10.45	47.03	114.24	215.07

表 3: 三角連鎖問題の実行時間（単位：秒）

n	1000	2000	4000	6000	8000
バイト数	2.3M	9.9M	43M	98M	176M
MiniSat	0.237	1.330	10.569	35.007	81.793
消費メモリ	22M	43MB	123M	256M	432M
mios	0.257	1.260	8.979	29.618	69.517
入力部	0.247	1.237	8.900	29.447	69.223
求解部	0.003	0.017	0.070	0.157	0.277
mios 1.0	1.30	5.76	27.54	71.30	143.44
toysat	2.31	10.38	46.80	112.84	211.61

リテラル数 n を変えてこれらの問題を作成し，実行した結果を表 1,2 3 に示す．なお，表中の MiniSat は移植元であるバージョン 1.14 を指す．また我々以外の手による Haskell で記述された最も高速なソルバーである toysolver[Sakai] での結果も参考として示す．

まず表 2 および表 3 における，ファイルサイズに対する実行時間増加の差異について説明する．この両者はリテラルの符号を除けばファイル内容は全く同一となる．にも関わらず消費メモリに違いが出ていることから，MiniSat においては節の組み立ておよび登録以前に入力時 *1 の前処理がなされていると思われる．入出力部は完全な移植ができていない部分であるため，ソルバー間の大きな差になったものと考えられる．前処理

*1 入力は節の登録，監視リテラルリストへの追加までを指し，求解とは最初の `simplifyDB` の呼出しを含む `solve` 関数以降の処理を指すものとする．

がなされない三角連鎖問題では両ソルバーの差は小さい（むしろよいものになっている）.

なお，mios-1.0 ではモナドを用いた合成可能な構文解析器である Parsec ライブラリ [Leijen01] を用いて構文解析を行っていたが，mios-1.1 ではストリーム型入出力パッケージを含むいくつかの選択肢の中で実装してもっともよい結果となった，ファイルを読み込むことで得られるバイト配列から節を表す配列へ（リストを経由せずに）直接代入を行うように書き直した．その結果，三角連鎖問題においては MiniSat と mios の差は解消しているが，全ての場合で差を埋めるものにはなっていない．

表 1 に示した単連鎖 (3) 問題においても入力に要する割合が大きい．特に同じく Haskell 言語による toysolver よりも遅いものとなっている．詳細の分析と改善は今後の課題である．一方，求解部では有効精度が低いものの最大で 2 倍程度の速度差にとどまり，「ghc は gcc に比べて 1 から 0.1 倍の性能を持つ」という Haskell コンパイラの性能に関する研究報告 [Petersen13] から予測される範囲内での性能を得ることができていることが分かる．

以上のことから，両ソルバーの実行時間の差異は主に入力部の処理の違いによるところが大きいこと，ただし，前処理が使用できないような状況では大きな差はないこと，ソルバー内部での基本処理に関してはせいぜい 3 倍程度の速度差に抑えることができていることがわかった．

4.2 ヒューリスティクス移植の検証

次にヒューリスティクス（具体的には VSIDS, first UIP を用いた節学習，学習節の管理）の移植の妥当性についての評価結果を示す．

4.2.1 3-SAT 問題

まず，問題規模に対する計算量の変化を見るために [Satlib] から入手できる，変数数が 100 から 250 の 3-SAT 問題を実行した．結果を図 1 に示す．横軸は変数数，縦軸はその変数数からなる無作為に抽出した 100 問の問題を全て解くのに必要な実行時間（秒）の対数表示である．古いソルバーのデータは [檜崎 16] より再掲した．

計測範囲の全域において，toysolver および mios-1.0 に対して大幅な速度の改善（2〜10 倍の高速化）を達成できている．実行時間を変数数に対する指数関数として近似し，最小二乗法により次数を求めると以下の値を得る．

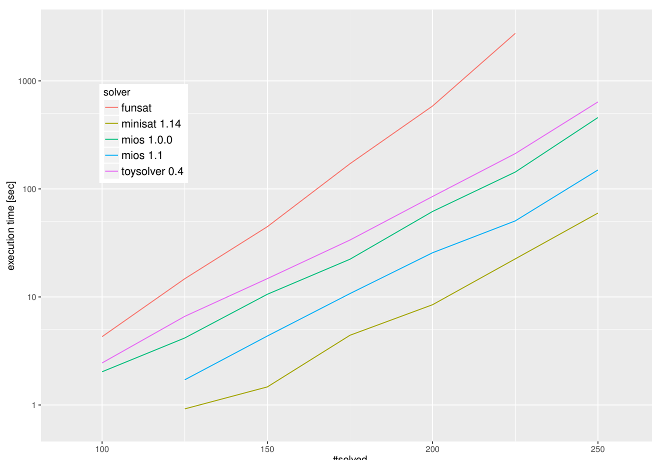

図 1: 3-SAT 問題での実行時間

MiniSat	mios-1.1	toysolver	mios-1.0
0.0362	0.0365	0.0375	0.0395

MiniSat に対する計算量の次数の違いはほぼ解消されている．また，MiniSat に対する速度比は最大で 2.9，平均で 2.5 となった．3-SAT 問題はファイルサイズに対して実行時間が大きいため，計算時間はほぼ全て求解部によるものと考えることができる．従って，4.1 節での結論と同様に 3 倍程度の速度低下に抑えることができていることがわかった．

4.2.2 SAT-Race 2015 ベンチマーク問題

次に SAT-Race 2015 の競技会で用いられた問題 [SAT-Race] を用いて，問題の規模が大きく，かつ現実的な問題に対する評価を行った．SAT-Race 2015 で提供される問題は約 300 問で構成されているため，まず最新版である MiniSat-2.2 が 1000 秒で解ける 131 問に絞り込み，それを各ソルバーで 1200 秒の時間制約下で実行した．結果を図 3 に示す．横軸は解けた問題数，縦軸はその問題の実行時間の対数表示である．

MiniSat-1.14 は 103 問を解くのに対し，mios-1.1 は 69 問を解いている．この結果は Haskell 言語によるソルバーの中では最もよいものであり，大きな改善である．また，mios-1.0 に対しても実行時間の傾向として 10 倍近い改善が見られ，忠実な移植の必要性を示すものとなった．図中の “minisat * 10” で示す線は MiniSat の実行時間を 10 倍したものである．よく近似できていることから，MiniSat に対する速度比は 10 倍と言えるであろう．この値は 4.1，4.2.1 節で得られたものよりはかなり悪い値となっている．

この速度低下が意図せぬヒューリスティクスの実装の不備によるものか，コンパイラの差などの不可避なものによるものかを評価するため，両ソルバーが共に解けた問題におけるバックトラック回数を比較した（両者とも乱数を使用するため両ソルバーの挙動の完全な一致は困難である）．両ソルバーのバックトラック回数の比の対数のヒストグラムを図 2 に示す．正の値は mios の方がバックトラック回数が多いことを，0 は同一であることを意味する．平均値は 0.36 であり，バックトラック回数は 1.43 倍の増加であることから，ヒューリスティクスに違いがあること，それだけでは速度低下を十分に説明できないことがわかった．

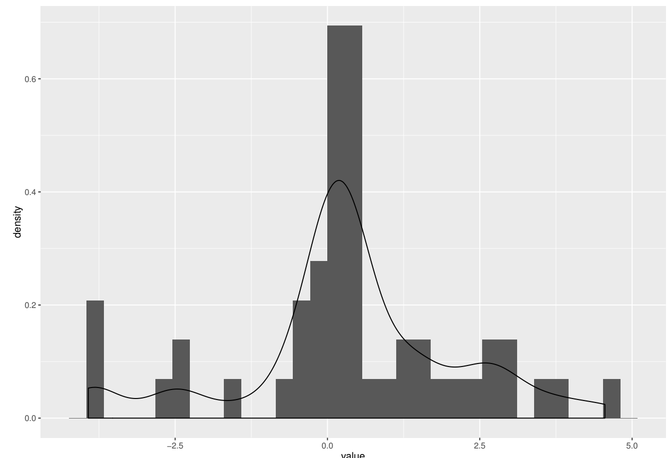

図 2: バックトラック回数の相対値

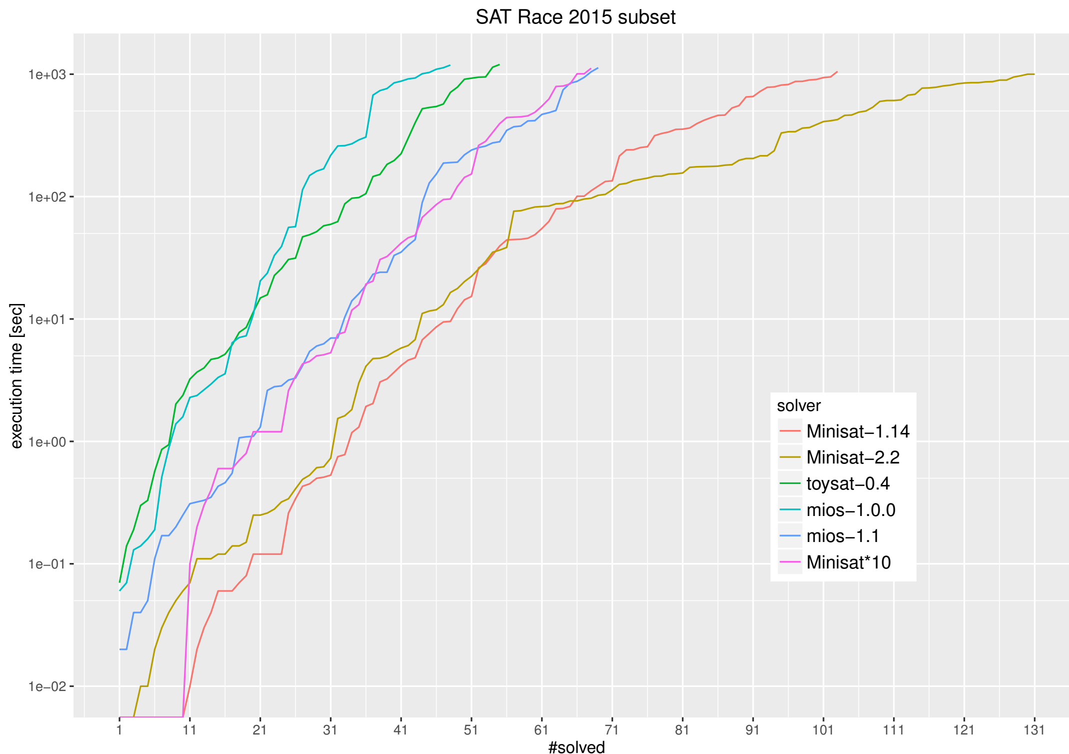

図 3: SAT-Race 2015 ベンチマーク問題の実行結果

5. まとめと今後の課題

今回の研究では Haskell 言語の特徴に留意して可能な限り忠実な移植を行うことで，以前のソルバーが達成できなかった性能を得た．比較的規模の小さな問題では同等の C++言語によるソルバーに対して速度低下をほぼ 3 倍程度に抑えることができている．一方で，規模の大きな現実的な問題に対しては 10 倍程度の速度低下となっている．この速度差が発生する原因は解明できていないため，今後の検討が必要である．

10 倍遅いものを実用的なソルバーと主張することは困難であるが，既に問題によっては MiniSat よりも速く解ける場合がある．また，言語の違いによる定数倍の速度差は使用するヒューリスティクスの差よりも小さいとも言える（例えば MiniSat-2.2 と MiniSat-1.14 との解けた問題数の差は mios と MiniSat-1.14 での差とほぼ等しい）．4.2.2 節で見られた速度低下の原因を解決し，より新しいアルゴリズムの研究成果を導入することで Haskell 言語による SAT ソルバーは将来的には十分実用的なものになりうると考えている．

今後は，残された課題を解決し，より高速な SAT ソルバーを実現することで Haskell 言語での SAT ソルバーの応用・研究の拡大を目指す予定である．

参考文献

[Audemard09] Gilles Audemard and Laurent Simon, “Predicting Learnt Clauses Quality in Modern SAT Solvers.” IJCAI. vol. 9. pp.399-404, 2009.

[Biere09] Armin Biere *et. al*, *Handbook of Satisfiability*, IOS Press, 2009.

[Bueno] Denis Bueno, `https://hackage.haskell.org/package/funsat`

[Een03] 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.

[Haskell] `https://www.haskell.org/`

[Leijen01] Daan Leijen and Erik Meijer. “Parsec: A practical parser library,” *Electronic Notes in Theoretical Computer Science*, vol.41, no.1, pp.1-20, 2001.

[Petersen13] Leaf Petersen *et. al*, “Measuring the Haskell Gap,” in *The 25th Symposium on Implementation and Application of Functional Languages*, 2013.

[Sakai] Masaki Sakai, `https://github.com/msakai/toysolver`

[Satlib] `http://www.cs.ubc.ca/~hoos/SATLIB/ benchm.html`

[SAT-Race] `http://baldur.iti.kit.edu/sat-race-2015/downloads/sr15bench.zip`

[鍋島 10] 鍋島英知，宋剛秀，高速 SAT ソルバーの原理，人工知能学会誌，25 巻 1 号, pp.68–76, 2010-01.

[楢崎 14] 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.

[楢崎 16] 楢崎修二，Minisat との比較による Haskell SAT ソルバーの高速化，IPSJ2016, 情報処理学会，4A-05, 2016-03.