関係的仕様からの関数型プログラム合成

中尾 收　佐竹 佑規　海野 広志

本研究では, 複数の再帰関数の入出力に関する関係的仕様と一部の関数の定義が与えられたときに, 残りの関数の定義を関係的仕様を満たすように自動的に合成する手法を提案する. 関係的仕様の具体例として, 等価性, 非干渉性, 単調性, 冪等性, 逆関数の存在性が挙げられる. したがって提案手法を用いると, 融合, 組化, 逆変換といった異なるプログラム変換を同じ枠組みで行うことができ, 変換が成功した場合にその正しさの証明も出力できる. 本研究では, [13] で提案された帰納的定理証明とホーン節制約解消に基づく関係的検証法を, 定義が不完全な関数を扱えるように拡張することで関係的合成を実現する. 具体的には, 定義が不完全な関数の入出力関係を表した述語変数が満たすべきホーン節制約を生成し, アブダクションによって解を求め, そこから関数定義を抽出する. 本研究ではさらに, 提案手法に基づいた合成器の実装および予備実験を行った.

1　はじめに

プログラム合成とは, 入出力例や自然言語による仕様, 形式的な仕様が与えられたときに, それらを満たすプログラムを自動的に生成する技術である [7]. 特に, 形式仕様からのプログラム合成が成功した場合は, 出力されたプログラムが仕様を満たすことが保証されるため, プログラム合成は, プログラム検証を不完全な定義を許すように一般化したものと捉えることができる. 特に形式仕様から関数型プログラムを合成する手法には, 合成問題を小さく単純なものに分解して効率的に解く Leon [8] や, 仕様を表現するためにリファインメント型を用いる Synquid [9], SMT ソルバ CVC4 のモデル生成機能を利用する合成手法 [10] が存在する.

本研究の目標は, 形式的な関係的仕様からの関数型プログラムの自動合成である. 関係的仕様とは, 複数のプログラムの間に成り立つ関係を表現した仕様のことである. 関係的仕様の具体例として, 等価性, 非干渉性, 単調性, 冪等性, 逆関数の存在性などが挙げられる. 関係的仕様からのプログラム合成の特に有用な応用例として, 融合 [5][15] や, 組化 [2][3], 逆変換 [4] といったプログラム変換が考えられる. 一般にプログラム変換は, 変換前のプログラムと変換前後のプログラムが満たすべき関係的仕様から, 変換後のプログラムを合成する問題として捉えることができる. つまり, 関係的仕様からのプログラム合成を実現すれば, 様々なプログラム変換を統一的に扱うことができ, さらに変換が正しいことの証明を出力することができるようになる. そのようなプログラム合成の具体例として, リストの偶数番目の要素を取り出す関数 even とリストの各要素の総和を計算する関数 sum から, リストの偶数番目の要素の総和を計算する関数 sum_even を合成する問題を図 1 に示す. ここで, リストの要素の順番は先頭から 0 番目, 1 番目と数える. sum と even を用いてリストの偶数番目の要素の合計を計算するにはリストの走査をそれぞれ 1 回ずつの合計 2 回行う必要があるが, 合成される sum_even は同等の計算を 1 度の走査によってまとめて行うように効率化される. このようなプログラム変換を融合変換という [2][3]. 図 1 で, "??" は合成によって自動的に生成したい部

Functional Program Synthesis from Relational Specifications. This is an unrefereed paper. Copyrights belong to the Authors.

Shu Nakao, Yuki Satake, Hiroshi Unno, 筑波大学システム情報工学研究科, Graduate School of Systems and Information Engneering, University of Tsukuba.

```
type list = Nil | Cons of int * list
let rec even x =
  match x with
  | Nil -> Nil
  | Cons(u, Nil) -> Cons(u, Nil)
  | Cons(u, Cons(_, us))
      -> Cons(u, even us)
let rec sum x =
  match x with
  | Nil -> 0
  | Cons(u, us) -> u + sum us
let rec sum_even = ??
let main x =
  assert (sum (even x) = sum_even x)
```

図 1　sum_even の合成問題

```
let rec sum_even x =
  match x with
  | Nil -> 0
  | Cons (u, Nil) -> u
  | Cons (u, Cons(_, us))
      -> u + sum_even us
```

図 2　sum_even の合成問題の解

分を表す. この例では, main 関数中の assert 式が失敗しないこと, つまり sum (even x) = sum_even x が false とならないこととして even, sum, sum_even の間の関係的仕様を与えている. このプログラム合成問題の解の 1 つを図 2 に示す.

しかし, 既存の関数型プログラム合成手法 Leon [8], Synquid [9], CVC4 [10] では, このような関係的仕様からのプログラム合成を十分に行うことはできなかった. Leon は合成問題を, 関数の入出力の間の関係を表す論理式の制約によって表現する. Leon はこの問題をより小さい部分問題に分解することで効率的なプログラム合成を実現している. しかし, 関係的仕様を分解するためには制約に出現する関数の定義を展開しなければならないが, 複雑な仕様では, 制約に出現する関数の定義を展開することができない場合がある. したがって, Leon の手法では関係的仕様からのプログラム合成は難しい. Synquid は, 合成する関数の仕様をリファインメント型によって与え, プログラムの型付けがその部分プログラムの型付け結果を用いてモジュラーに行われることを利用し, 型主導のプログラム探索を行うことで, 効率的なプログラム合成を実現している. 関係的仕様からの合成のためにはリファインメント型に出現する関数の定義を展開する必要があるが, Leon と同様に, 関数の定義を展開できない場合があるため関係的仕様からのプログラム合成は難しい. CVC4 は, 合成問題の形式に応じてそれに合った合成手法の適用を行うが, 関係的仕様からの合成のような, 複数の関数が出現するような形式の問題に有効な手法は我々の知る限り提案されていない.

そこで我々は, 関係的仕様からの関数型プログラム合成が可能な新しい手法を提案する. 提案手法は, 帰納的定理証明およびホーン節制約解消に基づいた関係的仕様検証法 [13] を, 不完全な定義の関数を許すように一般化することによってプログラム合成を可能にしたものであり, 重要な点は, (1) ホーン節制約を扱うことと, (2) 帰納法を用いることの 2 つである. それぞれについて説明する.

(1) ホーン節制約とは, (制約付き) ホーン節によって表された述語変数上の制約であり, 命令型, 関数型, 並行型といった様々なパラダイムのプログラム検証問題が, ホーン節制約の解消問題に帰着できることが知られている [11][12][14][6]. ホーン節制約に出現する述語変数は, 各プログラムポイントにおける帰納的不変条件を表しており, 制約を満たす述語変数の解の存在を示すことができれば, プログラムが仕様を満たすことが保証される. また, ホーン節制約は例外や代数データ型, 高階関数のような高度な言語機能や, 非決定性, 非停止性といった副作用を自然に扱うことができる. したがって, ホーン節制約は様々なパラダイムのプログラム検証問題を統一的に扱うための中間言語と捉えることができる. 同様に, 本研究では様々なパラダイムのプログラム合成問題を, ホーン節制約解消問題を一般化して定義が不完全な述語変数を許すようにしたホーン節制約アブダクション問題 (Horn Constriant Abduction Problem, HCAP) に帰着することによって, 様々なパラダイムのプログラム合成問題を統一的に扱えるようにする. 本論文では特に, 一階の決定的関数型プログラムを合成する手法を具体的に与える.

(2) 帰納法を用いる利点は, 関係的仕様を扱うことが

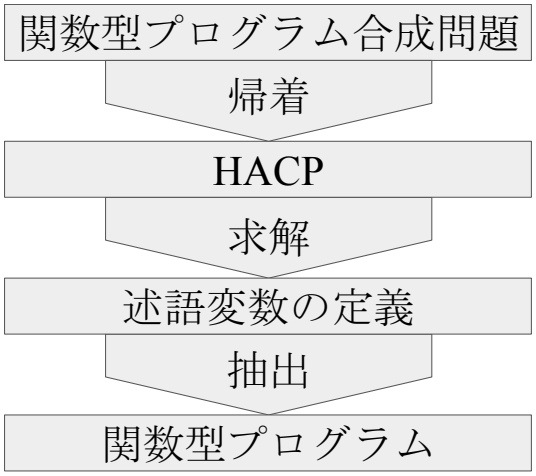

図 3　提案手法による合成の流れ

できることである. 帰納法を用いると, 複数の関数の間の相互不変条件を帰納法の仮定として表現し, 証明に用いることができるため, 複数の再帰関数の定義を同時に解析することが要求されるような関係的仕様の検証が可能となる [13]. 提案手法は, [13] で提案された, 帰納的定理証明とホーン節制約解消に基づくプログラム検証法を拡張してプログラム合成に用いることにより, 関係的仕様からのプログラム合成を実現する.

提案手法によるプログラム合成の流れを図 3 に示す. 提案手法はまず, プログラム合成問題を HCAP に帰着する. ここで, HCAP の解とは, 不完全な定義を持つ関数に対応する述語を定義するホーン節の集合で関係的仕様を満たすもののことである. そのような解を求めるため, 次に, 定義が不完全な述語の帰納的定義のテンプレートを生成し, テンプレートのパラメータが満たすべき制約を生成・解消する. 最後に, HCAP の解から一階の決定的関数型プログラムを抽出し, プログラム合成問題の解として出力する.

本研究ではさらに, 提案手法に基づいた, 関数型言語 OCaml のためのプログラム合成器の実装を行い, 新たに作成した関係的仕様からのプログラム合成問題のベンチマークセットを用いて評価実験を行なった.

本論文は以下のように構成される. 2 節では, sum_even の合成問題を例にとり, 提案手法の処理の流れを説明する. 3 節ではプログラム合成問題と HCAP を形式化し, プログラム合成問題から HCAP への帰着法を示す. 4 節では HCAP の求解法を示す. 5 節では, HCAP の解からプログラムを抽出する手法を示す. 6 節では実験の結果を示し, 考察を行う. 最後に, 7 節では結論を述べる.

2　提案手法の概要

ここでは, 合成問題の具体例として図 1 で示した sum_even の合成問題を用いて提案手法によるプログラム合成の流れを簡単に説明する.

2.1　合成問題の HCAP への帰着

図 3 における, 関数型プログラム合成問題から HCAP への帰着を行う部分を説明する. [12] で提案されたプログラム検証問題からホーン節制約を生成する手法を, 不完全な定義を持つ関数を許すように拡張して用いることで, 合成問題からホーン節制約を生成する. sum_even の合成問題からは以下のホーン節制約 $\mathcal{H}_{sum\_even}$ が生成される.

$$\left\{\begin{array}{l} P(\mathrm{Nil},\mathrm{Nil}),\ P(\mathrm{Cons}(u,\mathrm{Nil}),\mathrm{Nil}),\\ P(\mathrm{Cons}(u_1,\mathrm{Cons}(u_2,us)),\mathrm{Cons}(v,vs)) \Leftarrow\\ \quad P(us,vs)\wedge v=u_1,\\ Q(\mathrm{Nil},0),\ Q(\mathrm{Cons}(u,us),r'+u) \Leftarrow Q(us,r'),\\ \quad \bot \Leftarrow P(x,y)\wedge Q(y,r_1)\wedge R(x,r_2)\wedge r_1\neq r_2 \end{array}\right\}$$

述語変数 P, Q, R はそれぞれ関数 even, sum, sum_even の引数と返り値の間の帰納的不変条件を表す. R は不完全な定義の関数 sum_even に対応する述語変数であるため, R を定義するホーン節制約は生成されていない. それぞれの述語の引数のうち, 最後のものは関数の返り値に対応しており, 他の引数は関数の引数に対応している. ホーン節 $P(\mathrm{Nil},\mathrm{Nil})$ は関数 even の定義中の match 式の最初のケースから生成され, even の引数が Nil であるときに Nil が返されることを意味する. 対して $P(\mathrm{Cons}(u_1,\mathrm{Cons}(u_2,\mathrm{Nil})),\mathrm{Cons}(v,vs)) \Leftarrow P(us,vs)\wedge v=u_1$ は match 式の 3 番目のケースから生成され, even の引数が $\mathrm{Cons}(u_1,\mathrm{Cons}(u_2,us))$ の形をしているときは再帰呼び出し even us を行い, その結果を vs としたときに $\mathrm{Cons}(u_1,vs)$ を返すことを意味する. 最下行のホーン節 $\bot \Leftarrow P(x,y)\wedge Q(y,r_1)\wedge R(x,t_2)\wedge r_1\neq r_2$ は assert 式

から生成されており, 条件式 sum (even x) = sum_even x が false とならないための制約を表す.

このように生成されたホーン節制約と, 不完全な定義の関数に対応する述語変数の集合を入力として, それらの述語変数の定義をホーン節の集合の形で求める問題が HCAP である. この例では, HCAP は $\mathcal{H}_{sum\_even}$ を満たすように R を定義するホーン節の集合を求める問題となる.

ただし合成の目的においては, 解が制約を満たすだけでなく, 合成されるプログラムが望ましい性質をもつように解を求める必要がある. 本研究で合成するプログラムに望まれる性質は, 停止性が保証される入力の領域が広いことと決定的であることである. 例えば, $\mathcal{H}_{sum\_even}$ を満たす R を定義するホーン節の集合として自明な解 $\emptyset$ を即座に求めることができるが, $\emptyset$ は無限ループによって停止しない無意味なプログラムを表しており, 合成の目的においては望ましくない. そこで, 解の優劣を表す半順序関係を与え, 極大解を求めることによって望ましい性質を持った解を得る. sum_even に関する HCAP の極大解を以下に示す.

$$\left\{ \begin{array}{l} R(\mathrm{Nil}, 0), \quad R(\mathrm{Cons}(u, \mathrm{Nil}), u), \\ R(\mathrm{Cons}(u_1, \mathrm{Cons}(u_2, x')), r' + u_1) \Leftarrow R(x', r') \end{array} \right\}$$

この解は, 図 2 で示した, 任意の入力に対して停止する決定的なプログラムを表す. HCAP の詳細な定義は 3 節にて述べる.

2.2 HCAP の求解

図 3 における, HCAP の求解法について説明する. HCAP の解空間は膨大であるため, 提案手法は解のテンプレートを用いることによって探索空間を限定し, テンプレートのパラメータである未知論理式を決定することによって解を求める. つまり, テンプレートによって解の形を制限することで, 合成される関数定義に出現する補助関数の呼び出しや再帰呼び出しのパターンを制限している. したがって, テンプレートを適切に設定することで目的のプログラム変換が, 関係的仕様からのプログラム合成として実現される. sum_even の合成問題から帰着された HCAP の解を求めるために使用する解のテンプレートの一例を以下に示す.

$$\left\{ \begin{array}{l} R(x, r) \Leftarrow \nu_{base}(x, r), \\ R(x, r) \Leftarrow R(x', r') \wedge \nu_{ind}(x, r, x', r') \end{array} \right\}$$

テンプレート中の ν は未知論理式を表す. テンプレートの 1 行目は合成する関数 sum_even のベースケースに対応し, 2 行目は再帰呼び出しを行うインダクションケースに対応する.

テンプレートを追加した制約は関係的仕様を満たさなければならないので, 未知論理式を許すように [13] を拡張したホーン節制約解消法を用いて制約を解きながら未知論理式が満たすべき制約を集める. [13] はホーン節制約の解が存在することの証拠として帰納的定理証明システムの証明木を構築するので, 未知論理式を含むホーン節制約のための証明木を構築させ, その証明木が実際に解の存在性を含意するために未知論理式が満たすべき制約を証明木から抽出する. 上のテンプレートを利用して集めた未知論理式の制約を以下に示す.

$$\begin{array}{l} \bot \Leftarrow \nu_{base}(x, r_2) \wedge r_2 \neq 0 \wedge x = \mathrm{Nil}, \\ \bot \Leftarrow \nu_{ind}(x, r_2, x', r_2') \wedge r_2 \neq 0 \wedge x = \mathrm{Nil}, \\ \bot \Leftarrow \nu_{base}(x, r_2) \wedge r_2 \neq u \wedge x = \mathrm{Cons}(u, \mathrm{Nil}), \\ \bot \Leftarrow \nu_{ind}(x, r_2, x', r_2') \wedge r_2 \neq u \wedge x = \mathrm{Cons}(u, \mathrm{Nil}), \\ \bot \Leftarrow \nu_{base}(x, r_2) \wedge x = \mathrm{Cons}(u_1, \mathrm{Cons}(u_2, us)), \\ \bot \Leftarrow \nu_{ind}(x, r_2, x', r_2') \wedge r_2 \neq r_2' + u \wedge \\ \quad x = \mathrm{Cons}(u_1, \mathrm{Cons}(u_2, us)), \\ \bot \Leftarrow \nu_{ind}(x, r_2, x', r_2') \wedge x' \neq us \wedge \\ \quad x = \mathrm{Cons}(u_1, \mathrm{Cons}(u_2, us)) \end{array}$$

抽出された未知論理式の制約は, [1] で研究されたマルチアブダクション問題と捉えることができるため, 同文献で提案されている求解法を, 合成の目的で望ましい解を求めるように拡張して用いる. 上の未知論理式の制約の解を以下に示す.

$$\begin{array}{rcl} \nu_{base}(x, r) & \equiv & x = \mathrm{Nil} \wedge r = 0 \vee \\ & & x = \mathrm{Cons}(u_1, \mathrm{Nil}) \wedge r = u_1, \\ \nu_{ind}(x, r, x', r') & \equiv & x = \mathrm{Nil} \wedge r = 0 \vee \\ & & x = \mathrm{Cons}(u_1, \mathrm{Nil}) \wedge r = u_1 \vee \\ & & x = \mathrm{Cons}(u_1, \mathrm{Cons}(u_2, x')) \wedge \\ & & r = 1 + u_1 + r' \end{array}$$

この解をテンプレートに代入すると, 前節で示した HCAP の解が得られる.

2.3 HCAP の解からの関数型プログラム抽出

図 3 における, HCAP の解からの関数型プログラムの抽出法について説明する. 本手法は, HCAP の解から決定的な一階の関数型プログラムを抽出する. HCAP の解に現れるホーン節の一つ一つは, 関数の入力に応じた分岐に対応する. そのため, ホーン節ごとに対応する分岐とその分岐の条件を抽出する. sum_even の例では, ホーン節 $R(\mathrm{Nil}, 0)$ は sum_even の引数が Nil のときに 0 を返すことを意味し, ホーン節 $R(\mathrm{Cons}(u_1, \mathrm{Cons}(u_2, x')), r' + u_1) \Leftarrow R(x', r')$ は引数が $\mathrm{Cons}(u_1, \mathrm{Cons}(u_2, x')$ のとき再帰呼び出し sum_even x' を行い, その返り値を r' としたとき $r'+u_1$ を返すことを意味する. したがって, 2.2 節で示した HCAP の解からは図 2 で示した OCaml プログラムが抽出される.

3 プログラム合成から HCAP への帰着

3.1 ホーン節制約系

本研究で使用するホーン節制約系 (Horn Clause Constraint Sets, HCCSs) の構文を以下で定義する.

$$
\begin{array}{rrll}
\text{(HCCS)} & \mathcal{H} & ::= & \{hc_1, \ldots, hc_m\} \\
\text{(Horn Clause)} & hc & ::= & h \Leftarrow b \\
\text{(Head)} & h & ::= & P(\widetilde{t}) \mid \bot \\
\text{(Body)} & b & ::= & P_1(\widetilde{t_1}) \wedge \ldots \wedge P_m(\widetilde{t_m}) \wedge \phi \\
\text{(Formula)} & \phi & ::= & t_1 \leq t_2 \mid t_1 = t_2 \\
 & & & \mid is_C(t) \mid \top \mid \bot \mid \neg\phi \\
 & & & \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \\
\text{(Term)} & t & ::= & n \mid x \mid t_1 + t_2 \mid t_1 - t_2 \\
 & & & \mid C(\widetilde{t}) \mid s_{C,i}(t)
\end{array}
$$

簡単のため, ここでは量化子のない整数上の線形算術および代数データ型上の等式理論に限定して話を進める. P は述語変数, n は整数, x は変数, C は代数データ型のコンストラクタをそれぞれ表すメタ変数である. $\widetilde{t}$ は項列 $t_1, \ldots, t_m$ を, $s_{C,i}$ はコンストラクタ C の i 番目の引数を取得するアクセッサを, is_C は引数がコンストラクタ C で構成された代数データ型の値であるか判定する述語をそれぞれ表す. 述語変数 P およびコンストラクタ C のアリティをそれぞれ $ar(P), ar(C)$ と表す.

ホーン節制約系 $\mathcal{H}$ はホーン節の有限集合 $\{hc_1, \ldots, hc_m\}$ である. ホーン節とは肯定形の述語変数リテラルの数が 1 つ以下の節である. ホーン節 hc に出現する述語変数の集合を $pvs(hc)$ と表す. ホーン節制約系 $\mathcal{H}$ に関しては, 出現する述語変数の集合を $pvs(\mathcal{H}) = \bigcup_{hc \in \mathcal{H}} pvs(hc)$ と定義する. $\mathcal{H}$ 中の任意の異なるホーン節 hc_1, hc_2 は共通の自由変数を持たないとする. ホーン節のうち Head が $P(\widetilde{t})$ のものを確定節と呼び, $\mathcal{H}$ に含まれる確定節の集合を $def(\mathcal{H})$ と記述する. 同様に, Head が $\bot$ のものをゴール節と呼び, $\mathcal{H}$ 中のゴール節の集合を $goal(\mathcal{H})$ と記述する.

ホーン節制約系 $\mathcal{H}$ の述語解釈 ρ とは, $\mathcal{H}$ に出現する述語変数集合 $pvs(\mathcal{H})$ から $\mathbb{V}^{ar(P)}$ への写像である. ここで, $\mathbb{V}$ は整数と代数データ型の値すべてを含む意味領域を表す. ホーン節制約系 $\mathcal{H}$ の解とは, $\mathcal{H}$ に含まれる全てのホーン節 $hc \in \mathcal{H}$ について $\rho \models hc$ であるような述語解釈 ρ のことであり, このとき $\rho \models \mathcal{H}$ と書く.

3.2 ホーン節制約アブダクション問題

定義 3.1 (ホーン節制約アブダクション問題) ホーン節制約アブダクション問題 *(Horn Clause Abduction Problem, HCAP)* とは, ホーン節制約系 $\mathcal{H}$ と $\mathcal{H}$ に出現する述語変数の部分集合 $\mathcal{P} \subseteq pvs(\mathcal{H})$, 確定節の集合上の半順序 $\preceq$ の組 $(\mathcal{H}, \mathcal{P}, \preceq)$ として表される問題であり, その解とは, $\mathcal{P}$ 内の述語変数を定義する確定節集合 $\mathcal{D}$ のうち, ホーン節制約系 $\mathcal{H} \cup \mathcal{D}$ の解が存在し, $\preceq$ - 極大なものである. $\preceq$ - 極大な $\mathcal{D}$ とは, $\mathcal{D} \preceq \mathcal{D}'$ を満たす $\mathcal{D}$ でない $\mathcal{D}'$ が存在しないものである.

3.3 HCAP への帰着

プログラム合成問題からのホーン節制約生成は, プログラム検証問題からのホーン節制約生成法 [12] を, 不完全な定義の関数を扱えるように拡張して行う. 合成問題中に出現する "??" それぞれについてフレッシュな述語変数を生成する.

本研究の目標は, 広い領域で停止する決定的な一階の関数型プログラムを合成することである. そのため, 解を制約論理プログラムとしてとらえ, 入力を定めたときにその導出が決定的である方が大きく, また導出が停止するような入力の領域が広い方が大きいような

半順序 $\preceq_{det}$ を持つ HCAP へ帰着させる. ただし, 半順序 $\preceq_{det}$ が定義されるのは, 出現する述語変数が等しいものに限る.

4 HCAP の求解

$\preceq_{det}$ - 極大である HCAP の解を求めることは困難である. そこで, $\preceq_{det}$ - 極大な解を求める保証はないが, 多くの場合 $\preceq_{det}$ - 極大な解を求めることを目指したヒューリスティクスを用いた HCAP の求解法をこの節で提案する. $\preceq_{det}$ - 極大な解を厳密に求めるような求解法や, 任意の半順序の極大解を求めるような HCAP の求解法は, 今後の課題である.

HCAP の求解は, 以下に示す手順で行う.

1. 未知論理式を含む HCAP の解のテンプレートを作成する
2. 帰納的定理証明に基づくホーン節制約解消法 [13] によって証明木を構築し, 未知論理式に関する制約を集める
3. 集めた制約をマルチアブダクション問題として解くことで HCAP の解を得る
4. 3 で得た HCAP の解から, ヒューリスティックスを用いて $\preceq_{det}$ でより大きい解を得る

以降の節で, それぞれの手順について詳述する.

4.1 テンプレートの生成

未知論理式をパラメータにもつ HCAP の解のテンプレートを生成する. 未知論理式は, ホーン節に出現する述語変数適用の各引数の間の関係を表す. 2.2 節で示したテンプレート中の未知論理式 $\nu_{base}(x, r)$ は, Head の述語変数適用 $R(x, r)$ の引数 x, r の間の関係を表している. 同様に, 未知論理式 $\nu_{ind}(x, r, x', r')$ は, Head の述語変数適用 $R(x, r)$ と, Body の述語変数適用 $R(x', r')$ のそれぞれの引数 x, r, x', r' の間の関係を表している.

2 節では Body に出現する述語が Head と同じ述語変数を持つものに限ったテンプレートを説明したが, 他の述語の出現を許すこともできる. 複数の関数呼び出しを許す場合は, 対応する述語の組み合わせの数だけテンプレートを生成し, それぞれのホーン節に出現するすべての述語変数適用の引数の間の関係を表す未知論理式を作成すればよい.

4.2 未知論理式に関する制約の抽出

本手法は, 未知論理式に関する制約を集めるため, [13] の手法を拡張して用いる. 集められる制約は, Body に未知論理式が出現するゴール節の形をとる.

帰納的定理証明とホーン節制約に基づいたプログラム検証法 [13] は, 関係的仕様を表現するホーン節制約の解の存在を, 帰納的定理証明に基づいて判定する. 帰納的定理証明は, 判断を $\mathcal{D}; \Gamma; A; \phi \vdash h$ の形で記述し, 判断に対して推論規則を適用していき証明木を構築することで行う. $\mathcal{D}$ は述語変数を解釈するための確定節の集合, Γ は帰納法の仮定を表す論理式の集合, A は述語変数適用の集合, ϕ は述語変数適用を含まない論理式の集合, h はホーン節の Head である. 判断 $\mathcal{D}; \Gamma; A; \phi \vdash h$ は, $\mathcal{D}$ による述語の解釈と Γ の補題の元で, $\bigwedge A \wedge \phi \vdash h$ が妥当であることを表す. ホーン節制約 $\mathcal{H}$ 中のゴール節の集合 $goal(\mathcal{H}) = \{\bigwedge A_i \wedge \phi_i \Rightarrow \bot\}_{i=1}^{m}$ から判断 $def(\mathcal{H}); \emptyset; A_i; \phi_i \vdash \bot$ を生成することで, ホーン節制約解消問題を, 全ての判断について証明木を構築する定理証明問題に帰着させる.

本手法は, 未知論理式を含むホーン節制約を元に生成される判断に対して証明規則を適用していくことで証明木を構築し, 構築した証明木が解の存在性を含意するために必要な未知論理式の制約条件を抽出する. ただし, 判断内の Head h は $\bot$ に限られる.

sum_even の合成問題の例から生成される判断 J_1 を以下に示す.

$J_1 \equiv \mathcal{D}; \emptyset; \{P(x, y), Q(y, r_1), R(x, r_2)\}; r_1 \neq r_2 \vdash \bot$

4.2.1 推論規則

本手法で判断を導出するために用いる推論規則は, [13] の推論規則のうち, INDUCT 規則, UNFOLD 規則, APPLY$\bot$ 規則, VALID$\bot$ 規則の 4 つである. それぞれについて説明する. INDUCT 規則は判断中の A 内の述語変数適用の一つを選び, その述語変数適用の導出に関する帰納法を適用することで帰納法の仮定を作成し, Γ に追加する. UNFOLD 規則は判断中の A 内の述語変数適用 $P(\widetilde{t})$ を一つ選び, その述語変数適用の導出木を得るために最後に使われた規則 $P(\widetilde{t}) \Leftarrow \phi_i \wedge \bigwedge A_i$

を $\mathcal{D}$ から取り出し, 場合分けを行う. このとき場合分けの数だけ証明木は分岐する. 判断中の ϕ は $\phi \wedge \phi_i$ に, A は $A \cup A_i$ に更新される. APPLY$\bot$ 規則は Γ に含まれる帰納法の仮定を一つ選び, 対応する述語変数適用に適用する. VALID$\bot$ 規則は, 判断中の ϕ が $\phi \models \bot$ であるときに証明木の枝を閉じる.

4.2.2 証明木の推定戦略

未知論理式が含まれるホーン節制約には, [13] が採用している証明戦略をそのまま使用することはできない. そこで本手法では合成のために, HCAP の述語変数の集合 $\mathcal{P}$ に注目した以下の戦略を採用する.

1. 最初の判断 $\mathcal{D}; \emptyset; A_1; \phi_1 \vdash \bot$ の A_1 に含まれる述語変数適用のうち, その述語変数が $\mathcal{P}$ に含まれないもの全てについて, それぞれ 1 度ずつ UNFOLD 規則と INDUCT 規則を適用する.
2. UNFOLD 規則によって新たに A に追加された述語変数適用の, 入力に対応する代数データ型の引数について, ϕ からどのコンストラクタで構成されているかという情報が得られる場合, その述語変数適用に対してもう一度 UNFOLD 規則を適用する. これを繰り返す.
3. 判断 $\mathcal{D}; \Gamma; A; \phi \vdash \bot$ の A に含まれる述語変数適用のうち, その述語変数が $\mathcal{P}$ に含まれるもの全てについて, それぞれ 1 度だけ UNFOLD 規則を適用する.
4. UNFOLD 規則によって展開された確定節がすべて述語変数適用を含むような場合の枝に, APPLY$\bot$ 規則を適用する.
5. VALID$\bot$ 規則を適用し枝を閉じる.
6. 以上全ての手順において, 判断 $\mathcal{D}; \Gamma; A; \phi \vdash \bot$ 中の論理式 ϕ が更新された際, $\phi \models \bot$ であるなら即座に VALID$\bot$ 規則を適用し枝を閉じる.

この戦略を, sum_even の合成問題から生成された判断 J_1 に適用して得られる証明木を図 4 に示す.

戦略に沿って順に説明する. 最初の判断 J_1 に対し, 手順 1 に従い, 述語変数適用 $P(x, y), Q(y, r_1)$ について, INDUCT 規則と UNFOLD 規則をそれぞれ適用する. 述語変数適用 $P(x, y)$ に関して INDUCT 規則を適用することによって, 帰納法の仮定 $P(x, y) \wedge Q(y, r_1) \wedge R(x, r_2) \Rightarrow r_1 = r_2$ を J_1 に追加した判断 J_2 を生成している. この仮定は, 以降の証明規則の適用により新たに追加された述語変数適用 $P(x', y')$ について, y' が等しい述語変数適用 $Q(y', r_1')$ と, x' が等しい述語変数適用 $R(x', r_2')$ が存在する場合に APPLY$\bot$ 規則によって適用することができる. この仮定を適用すると, 判断中の論理式 ϕ を $\phi \wedge r_1' = r_2'$ に更新することができる. J_2 からは, 述語変数適用 $P(x, y)$ に注目して UNFOLD 規則を適用する. $\mathcal{D}$ 中に P を定義する確定節は 3 つ存在するため, UNFOLD 規則を適用すると証明木が 3 つに分岐する. 分岐の先でも Q について同様に, INDUCT 規則を適用して帰納法の仮定を追加したのち, UNFOLD 規則により証明木を分岐させる.

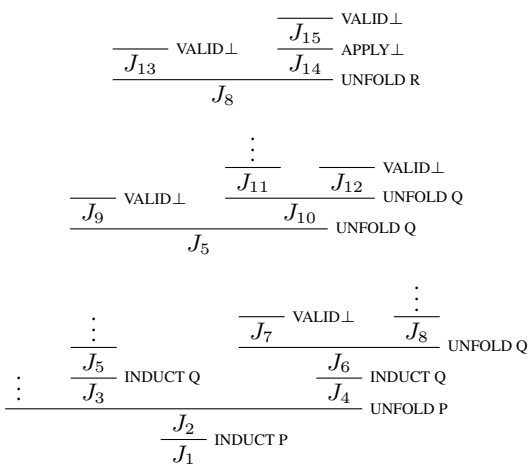

図 4 sum_even の合成問題で構築される証明木

UNFOLD する確定節の組み合わせによっては, 判断中の論理式 ϕ が充足不可能となる場合が存在する. 判断 J_9 中の論理式 ϕ_9 について注目すると, $\phi_9 \Rightarrow x = \mathrm{Cons}(u_1, \mathrm{Nil}) \wedge x = \mathrm{Nil}$ である. しかし, そのような x は存在しないため, $\phi_9 \vdash \bot$ であることから VALID$\bot$ 規則によって枝を閉じることができる.

次に手順 2 を適用する.UNFOLD 規則によって $P(\mathrm{Cons}(u, \mathrm{Nil}), \mathrm{Cons}(u, \mathrm{Nil}))$, $Q(\mathrm{Cons}(u, us), r' + u) \Leftarrow Q(us, r')$ の確定節に関する分岐を行なった判断 J_5 について注目して説明する. 新しく追加された述語変数適用は $Q(us, r')$ である. この判断中の論理式 ϕ_5 について, $\phi_5 \Rightarrow us = \mathrm{Nil}$ であるため, $Q(us, r')$

に注目して UNFOLD 規則を適用する.

手順 2 が完了したならば, 手順 3 に従い不完全な定義の関数に対応する述語変数適用 $R(x, r_2)$ に関する UNFOLD 規則の適用を行う. 判断 J_8 に注目して説明する. $R(x, r_2)$ を UNFOLD することにより, 解のテンプレート $R(x, r) \Leftarrow \nu_{base}(x, r)$, $R(x, r) \Leftarrow R(x', r') \wedge \nu_{ind}(x, r, x', r')$ 中のそれぞれの未知論理式 ν が判断中の ϕ に追加される.

手順 3 が完了したならば, UNFOLD 規則による分岐が全て述語変数適用を判断に追加するような枝について, APPLY$\bot$ 規則を適用する. 判断 J_{14} がその条件に当てはまる枝であるため, それに注目して説明する. 判断 J_{14} 内の述語変数適用は, $P(x, y), Q(y, r_1), R(x, r_2), P(us, vs), Q(vs, r_1'), R(x', r_2')$ である. このとき, 帰納法の仮定 $P(x, y) \wedge Q(y, r_1) \wedge R(x, r_2) \Rightarrow r_1 = r_2$ を述語変数適用の組 $P(us, vs), Q(vs, r_1'), R(x', r_2')$ に適用して, 判断中の論理式 ϕ_{14} を $r_1' = r_2' \wedge \phi_{14}$ に更新する.

以上の手順を全て終えて, 判断中の論理式 ϕ に未知論理式 ν を含むものは VALID 規則を適用し証明木の枝を閉じる.

この戦略により, 定義済みの関数と似た再帰呼び出しを行う関数を合成することができる. 複雑な再帰呼び出しを行う関数の合成はこの戦略だけでは不十分であるため, より洗練された戦略を発見することが将来の課題である.

4.2.3 証明木からの未知論理式の制約抽出

証明木の構築において, VALID$\bot$, APPLY$\bot$ 規則を適用する際に制約を抽出する.

判断 $\mathcal{D}; \Gamma; A; \phi \vdash \bot$ に対し VALID$\bot$ 規則を適用する条件に $\phi \models \bot$ という制約が含まれる. したがって ϕ 中に未知論理式が含まれていてかつ $\phi \models \bot$ が自明でない場合, $\phi \Rightarrow \bot$ が制約として抽出される.

判断 $\mathcal{D}; \Gamma; A; \phi \vdash \bot$ に対し APPLY$\bot$ 規則を適用する際, $\bigwedge A' \wedge \phi' \vdash h$ という形の帰納法の仮定が参照される. A' は仮定あるいは補題に必要な述語変数適用の集合であり, 述語変数適用に対応するものを A から適切に選出する. その際 A から選出される述語変数適用の中に不完全な定義の関数に対応する述語 P が含まれており, かつ A' 中で P の引数と共通の変数を持つ述語変数適用が存在する場合, その述語変数適用のために A から選出された述語変数適用の対応する引数とその引数が等しいという制約が抽出される.

図 4 で示した証明木の J_{14} に APPLY$\bot$ 規則を適用している部分では, 帰納法の仮定を適用するために必要な条件が $x' = us$ であるため, 制約 $\phi_{14} \Rightarrow x' = us$ が抽出される. また, 未知論理式を含む判断に対して VALID$\bot$ 規則を適用している部分から, それぞれ未知論理式が満たすべき制約が抽出される. それらを全て集めると, 2.2 節で示した制約となる.

4.3 マルチアブダクションによる求解

証明木から集めたそれぞれの制約について未知論理式の解を求め, 最後に全ての解の論理積を取ったものを解のテンプレートに代入したものを HCAP の解とする. 未知論理式の解を求めるために, [1] にて紹介されているマルチアブダクション問題の最弱解を求める手法を, 証明木から集めた制約の最弱解を求めるように拡張して適用する. マルチアブダクション問題とは, 複数種類の述語変数適用が複数回出現する制約を満足する述語変数の解を求める問題である. ここで最弱解とは, その解より論理的に弱い解が存在しないものをいう.

[1] の手法は, 与えられた制約を満足する述語の最弱解をモデル生成と量化子除去 (Quantifier Elimination, 以下 QE) を用いて求める. その手法は以下に示す 3 つのステップに分かれている.

1. モデルによって生成した述語変数の初期解を用いて, 制約を述語変数の種類ごとに分解する
2. 分解した制約を, さらに述語変数適用の出現ごとに分解する
3. 述語変数の初期解を生成し, QE によってその解を弱める

この手法には制約の分解によっては, 解を無限に弱め続けるケースに陥ることや, 各々の述語は最弱だが全体では最弱でないような局所解を出力することがあるという問題があった. この問題を解決するために, 分解を行う際にまず論理式の形による分解を試みることで, 問題が起きるような分解をなるべく防ぐような拡張を行った.

4.2.3 節にて抽出された sum_even の問題の制約を解くと, 未知論理式の解は以下のようになる.

$$\begin{aligned}\nu_{base}(x,r) &\equiv x = \mathrm{Nil} \wedge r = 0 \vee \\ &\quad\ x = \mathrm{Cons}(u, \mathrm{Nil}) \wedge r = u, \\ \nu_{ind}(x,r,x',r') &\equiv x = \mathrm{Nil} \wedge r = 0 \vee \\ &\quad\ x = \mathrm{Cons}(u_1, \mathrm{Nil}) \wedge r = u \vee \\ &\quad\ x = \mathrm{Cons}(u_1, \mathrm{Cons}(u_2, x')) \wedge \\ &\quad\ r = u_1 + r'\end{aligned}$$

4.4 より大きい解へのアプローチ

マルチアブダクション問題を解くことによって未知論理式の解を求めることで得た HCAP の解は, 制約論理プログラムとして捉えたときに, 入力を定めたときの導出が複数存在するという意味で決定的ではない. そのため, 入力を定めたときの導出が一意に定まるように解の決定化を行うヒューリスティックスを述べる. HCAP の解の中の, 不完全な定義の関数に対応する述語変数を定義する各々の確定節について, 述語の入力変数の領域で重なるものが存在するとき, その 2 つの差集合をとって片方の入力変数の領域を狭めることで, 重なる領域が存在しないようにする. このとき, 再帰的な定義の確定節の領域を狭めるようにする.

このヒューリスティックスを適用することで, 制約論理プログラムの意味で, 入力を定めたときにその導出が一意に定まるように決定化している. しかし, 出力のモデルが複数存在するような場合に出力を一意に定めることや, 停止しないような導出を避けるように決定化することはできない.

4.3 節で求めた sum_even の問題の解は以下のホーン節の集合である.

$$\left\{\begin{array}{l} R(\mathrm{Nil}, 0), \quad R(\mathrm{Cons}(u, \mathrm{Nil}), u), \\ R(\mathrm{Nil}, 0) \Leftarrow R(x', r'), \\ R(\mathrm{Cons}(u, \mathrm{Nil}), u) \Leftarrow R(x', r'), \\ R(\mathrm{Cons}(u_1, \mathrm{Cons}(u_2, x')), r' + u_1) \Leftarrow R(x', r') \end{array}\right\}$$

この解が表すプログラムは, 入力が Nil である場合と, $\mathrm{Cons}(u, \mathrm{Nil})$ の形をしている場合に当てはまる確定節が複数存在するため, 非決定的である. この解に対して決定化を行うと, 以下に示すような決定的なホーン節の集合となる.

$$\left\{\begin{array}{l} R(\mathrm{Nil}, 0), \quad R(\mathrm{Cons}(u, \mathrm{Nil}), u), \\ R(\mathrm{Cons}(u_1, \mathrm{Cons}(u_2, x')), r' + u_1) \Leftarrow R(x', r') \end{array}\right\}$$

Algorithm 1: extract

def extract$(P; \mathcal{D})$
(where $\mathcal{D} = \{P(\widetilde{x}, y) \Leftarrow b_i \mid 1 \le i \le m\}$,
$b_i = \bigwedge_j^{n_i} P_{ij}(\widetilde{x}_{ij}, y_{ij}) \wedge \phi_i$*)*
=

```
if QE(∃x̄̃.φ1) then
  extract_branch(λy.b1; x̃)
else if ...
else if QE(∃x̄̃.φn) then
  extract_branch(λy.bn; x̃)
else inf_loop()        // 無限ループ
```

5 HCAP の解からのプログラム抽出

HCAP の解は確定節の有限集合であるため, 制約論理プログラムと捉えることができる. ここでは, 制約論理プログラムから 1 階の決定的な関数の定義の抽出を試みる手法を述べる. 他のパラダイムのプログラムを抽出したい場合は, それに特化した抽出の手法を与える必要がある. 抽出法のバリエーションを増やすことは今後の課題である.

述語変数 P と, P を Head にもつ確定節の有限集合 $\mathcal{D}$ から, P に対応する関数を定義を抽出する手続き extract を Algorithm 1 に示す. $\mathcal{D}$ 中の確定節 $P(\widetilde{x}, y) \Leftarrow \bigwedge_j^{n_i} P_{ij}(\widetilde{x}_{ij}, y_{ij}) \wedge \phi_i$ は, P に対応する関数の入力に応じた分岐のうちの一つを表す. $\widetilde{x}$ は関数の入力に対応する変数, y は返り値に対応する変数である. QE$(\exists\overline{\widetilde{x}}.\phi_i)$ は, ϕ_i に出現する変数のうち $\widetilde{x}$ に含まれないものを存在量化した論理式の量化子除去を行った論理式を表す. QE によって, ϕ_i を入力変数 $\widetilde{x}$ の領域に射影したものが if 式の分岐条件となる. 分岐の本体の式は, 手続き extract_branch によって抽出される. どの分岐条件にも当てはまらない入力が存在する場合は, 無限ループを起こす分岐に到達する.

if 式の各分岐を抽出する手続き extract_branch の定

Algorithm 2: extract_branch

def extract_branch($\lambda y.(\bigwedge \emptyset \wedge \phi); \widetilde{u}$)=
  decide($\widetilde{u}; \lambda y.\phi$)
def extract_branch
($\lambda y.(\bigwedge_{i=1}^{n} P_i(\widetilde{x}_i, y_i) \wedge \phi); \widetilde{u}$)=
  let $\widetilde{t}$ = decide($\widetilde{u}; \lambda \widetilde{x}_1.\phi$) in
  let $\phi' = \phi[\widetilde{x}_1 \mapsto \widetilde{t}\,]$ in
  `let` y_1 = fun_of(P_1)($\widetilde{t}$) `in`
  extract_branch
  ($\bigwedge_{i=2}^{n} P_i(\widetilde{x}_i, y_i) \wedge \lambda y.\phi'; y_1, \widetilde{u}$)

Algorithm 3: decide

def decide($\widetilde{u}; \phi$)=
  ϵ // 空列
def decide($\widetilde{u}; \lambda x_1, \ldots, x_n.\phi$)=
  let ϕ' = QE($\exists \overline{x_1, \widetilde{u}}.\phi$) in
  let $\psi = \forall \widetilde{u}.((\exists x_1.\phi') \Rightarrow \phi'[x_1 \mapsto f(\widetilde{u})])$ in
  let M_f = model(ψ) in
  $M_f(\widetilde{u})$,
  decide($\widetilde{u}; \lambda x_2, \ldots, x_n.\phi[x_1 \mapsto M_f(\widetilde{u})]$)

義を Algorithm 2 に示す. extract_branch の入力は, 合成される式が満たすべき述語 $\lambda y.b$ と式に出現してよい自由変数の列 $\widetilde{u}$ である. b 中の述語変数適用 $P_i(\widetilde{x}_i, y_i)$ は, P_i に対応する関数の呼び出しの入力に対応する変数が $\widetilde{x}_i$, 返り値に対応する変数が y_i であることを表す. ただし, b 中の述語変数適用の各引数として渡される変数は互いに異なるものとする. $\lambda y.b$ に述語変数適用が現れない場合, つまり関数呼び出しを行わない式を抽出する場合は, 後述の手続き decide を用いる.1 つ以上の述語変数適用が現れる場合, 先頭の 1 つ $P_1(\widetilde{x}_1, y_1)$ に対応する関数呼び出しを作成し, 関数呼び出しの返り値に y_1 を let 束縛して残りの述語変数適用を再帰的に処理する. ここで, y_1 を束縛する let 式はプログラムの構文上の let 式であるのに対し, それ以外の let 式はアルゴリズムレベルの let 式なので, extract_branch が返す式に出現しないということに注意されたい. 関数呼び出しのために, 引数の列 $\widetilde{x}_1$ に対応する式の列 $\widetilde{t}$ も手続き decide によって決定する. そして, 論理式 ϕ 中の $\widetilde{x}_1$ を $\widetilde{t}$ で置き換えた論理式 ϕ' を得る. fun_of(P_1) は, 述語変数 P_1 に対応する関数を表す.

手続き decide の定義を Algorithm 3 に示す. decide の引数は, 合成される n 個の式に出現してよい自由変数の列 $\widetilde{u}$ と, n 個の式が満たすべき述語 $\lambda x_1, \ldots, x_n.\phi$ である. $n = 0$ の場合, decide は空列を返す. $n > 0$ の場合, まず x_1 に対応する式を合成し, 残りの $x_2, \ldots, x_n$ についても再帰的に合成する. 与えられた述語の論理式 ϕ には, $x_1, \ldots, x_n, \widetilde{u}$ の他にも自由変数が出現するため, QE によって $x_1, \widetilde{u}$ 以外の変数を除去した論理式 ϕ' を得る. x_1 に対応する式は, $\forall \widetilde{u}.((\exists x_1.\phi') \Rightarrow \phi'[x_1 \mapsto f(\widetilde{u})])$ を満たす関数 f のモデル M_f を SMT ソルバによって生成することにより合成する.

6 実装と実験

提案手法に基づいた関数型プログラム合成器のプロトタイプを, 関数型プログラム検証器 RCaml [12] を拡張することによって実装した. RCaml には帰納法に基づいたホーン節制約ソルバ [13] がすでに組み込まれていたので, それを拡張した.

本研究では, 独自に作成したベンチマークセットを用いて実装した合成器の評価実験を行なった. ベンチマークセットは本論文の付録に添付する. 表 6 に実験結果を示す. kind 列は合成問題としてどのような関係的仕様を与えたのかを示す. eq は等価性, inv は逆関数の存在性, incr は増加性を表す. 特に, sum_even は融合変換の問題である. result 列には合成に成功したものについては得られたプログラムを示し, 失敗したものについてはその原因を示す. wrong proof tree は証明木の探索に失敗していることを表す. stack overflow は, 合成の際にスタックオーバーフローが起きてしまったことを表す.

また, 関数型プログラム合成器 Leon [8] について, 今回作成したベンチマークセットと同等のものを入力として与え実験した. 表 6 に実験結果を示す. 実験では time out を 1 分に設定した. Leon は再帰的な定義を持つ関数の逆関数の合成問題 sub_acc を解けていな

表 1　提案手法の実験結果

problem	kind	result
mult_acc	eq	mult_acc x y a = if y = 0 then a else mult_acc x (y-1) (a+x)
mult	eq	mult x y = if y = 0 then 0 else x + mult x (y-1)
mult_int		mult x y = x * y
mult_dist	eq	wrong proof tree
sum_acc	eq	wrong proof tree
sub	inv	sub x y = x - y
sub_rec	inv	sub x y = if y = 0 then x else sub (x-1) (y-1)
pred	inv	pred x = x - 1
double1	eq	wrong proof tree
double2	eq	double x = if x = 0 then 0 else 2 + double (x - 1)
incr	incr	incr x = x + 1
max		max x y = if x $>$ y then x else y
sum_even	eq	sum_even l = match l with Nil $\rightarrow$ 0 $\mid$ Cons(u,Nil) $\rightarrow$ u $\mid$ Cons(u, Cons (_, us)) $\rightarrow$ u + sum_even us
sum_scs	eq	sum_scs l = match l with Nil $\rightarrow$ 0 $\mid$ Cons(u, us) $\rightarrow$ 1 + u + sum_scs us
tup_sum_even	eq	wrong proof tree
pred_dup	inv	stack overflow

い. また, sum_even の合成結果は, sum と even の 2 つの関数呼び出しを行なっているため, 融合変換になっていない.

7　まとめ

本論文では, 関係的仕様から関数型プログラムを合成するための帰納的定理証明およびホーン節制約に基づいた手法を提案した. また提案手法の実装とベンチマークを用いた評価実験を行い, 既存手法との比較を行った. 既存手法では難しかった, プログラム合成によるプログラム変換において有望な結果を得た.

参考文献

[1] Albarghouthi, A., Dillig, I., and Gurfinkel, A.: Maximal Specification Synthesis, *Proc. POPL '16*, ACM, 2016, pp. 789–801.

[2] Bird, R. S.: Using circular programs to eliminate multiple traversals of data, *Acta Informatica*, Vol. 21, No. 3(1984), pp. 239–250.

[3] Chin, W.-N.: Towards an Automated Tupling Strategy, *Proc. PEPM'93*, ACM, 1993, pp. 119–132.

[4] Dijkstra, E. W.: *Program Inversion*, Springer New York, 1982, pp. 351–354.

[5] Gill, A., Launchbury, J., and Peyton Jones, S. L.: A short cut to deforestation, *Proc. FPCA'93*, ACM, 1993, pp. 223–232.

[6] Grebenshchikov, S., Lopes, N. P., Popeea, C., and Rybalchenko, A.: Synthesizing Software Verifiers from Proof Rules, *Proc. PLDI'12*, ACM, 2012, pp. 405–416.

[7] Gulwani, S.: Dimensions in Program Synthesis, *Proc. PPDP '10*, ACM, 2010, pp. 13–24.

[8] Kneuss, E., Kuraj, I., Kuncak, V., and Suter, P.: Synthesis Modulo Recursive Functions, *Proc. OOPSLA '13*, ACM, 2013, pp. 407–426.

[9] Polikarpova, N., Kuraj, I., and Solar-Lezama, A.: Program Synthesis from Polymorphic Refinement Types, *Proc. PLDI '16*, ACM, 2016, pp. 522–538.

[10] Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., and Barrett, C.: Counterexample-Guided Quantifier Instantiation for Synthesis in SMT, *Proc. CAV'15*, 2015, pp. 198–216.

[11] Rondon, P., Kawaguchi, M., and Jhala, R.: Liquid Types, *Proc. PLDI'08*, ACM, 2008, pp. 159–169.

[12] Unno, H. and Kobayashi, N.: Dependent Type Inference with Interpolants, *Proc. PPDP'09*, ACM, 2009, pp. 277–288.

[13] Unno, H., Torii, S., and Sakamoto, H.: Automating Induction for Solving Horn Clauses, *Proc. CAV'17*, 2017, pp. 571–591.

[14] Vazou, N., Seidel, E. L., Jhala, R., Vytiniotis, D., and Peyton Jones, S. L.: Refinement Types for Haskell, *Proc. ICFP'14*, ACM, 2014, pp. 269–282.

[15] Wadler, P.: Deforestation: transforming programs to

表 2　Leon の実験結果

problem	result
mult_acc	mult_acc(x,y,a) = mult(x, y) + a
mult	mult(x,y) = mult_acc(x, y, 0)
mult_int	mult(x, y) = x * y
mult_dist	mult_dist(x, y, z) = mult(x, z) + mult(y, z)
sum_acc	sum_acc(x, a)= a + sum(x)
sub	sub(x,y) = x - y
sub_rec	time out
pred	pred(x) = x - 1
double1	double(x) = mult(x, 2)
double2	double(x) = mult(2, x)
incr	failed
max	max(x, y) = if (x $\geq$ y) x else y
sum_even	sum_even(l) = sum (even(l))
tup_sum_even	tup_sum_even(l) = (sum(l), even(l))
pred_dup	time out

eliminate trees, *Proc. ESOP'88*, LNCS, Vol. 300, 1988, pp. 344–358.

A　ベンチマーク

本研究で作成した合成器の実験に使用したベンチマークプログラムを以下に示す.

mult_acc

```
let rec mult x y =
  if y = 0 then 0 else y + mult x (y-1)
let rec mult_acc x y a = ??
let main x y a =
  assert(a + mult x y = mult_acc x y a)
```

mult

```
let rec mult x y = ??
let rec mult_acc x y a =
  if y = 0 then a
  else mult_acc x (y-1) (a+x)
let main x y a =
  assert(a + mult x y = mult_acc x y a)
```

mult_int

```
let rec mult x y = mult x y
let main x y =
        assert(x * y = mult x y)
```

mult_dist

```
let rec mult x y =
  if y = 0 then 0 else x + mult x (y-1)
let rec mult_dist x y z = ??
let main x y z =
  assert (mult x z + mult y z
    = mult_dist x y z)
```

sum_acc

```
let rec sum n =
  if n < 0 then n + sum (n + 1)
  else if n = 0 then 0
  else n + sum (n - 1)
let rec sum_acc n a = ??
let main n a =
  assert(a + sum n = sum_acc n a)
```

sub

```
let rec add x y = x + y
let rec sub x y = ??
let main x y =
  assert (add y (sub x y) = x)
```

sub_rec

```
let rec add x y =
  if x = 0 then y else 1 + add (x-1) y
let rec sub x y = ??
```

```
let main x y =
  assert (add y (sub x y) = x)
```

pred

```
let succ x = x + 1
let rec pred x = ??
let main x =
  assert (pred (succ x) = x)
```

double1

```
let rec mult x y =
  if y = 0 then 0 else x + mult x (y-1)
let rec double x = ??
let main x =
  assert (mult x 2 = double x)
```

double2

```
let rec mult x y =
  if y = 0 then 0 else x + mult x (y-1)
let rec double x = ??
let main x =
  assert (mult 2 x = double x)
```

mono

```
let rec mono x = ??
let main x y =
  if x > y then assert (mono x > mono y)
  else ()
```

incr

```
let rec incr x = ??
let main x =
  assert (incr x > x)
```

comm

```
let rec comm x y = ??
let main x y =
  assert(comm x y = comm y x)
```

max

```
let rec max x y = ??
let main x y =
  if x > y then
    assert (max x y = x)
  else
    assert (max x y = y)
```

sum_even

```
type list = Nil | Cons of int * list
let rec sum l =
  match l with
  | Nil -> 0
  | Cons(u, us) -> u + sum us
let rec even l =
  match l with
  | Nil -> Nil
  | Cons(u, us) -> Cons(u, us)
  | Cons(u1, Cons(u2, us))
    -> Cons(u1, even us)
let rec sum_even l = ??
let main l =
  assert (sum (even l) = sum_even l)
```

sum_scs

```
type list = Nil | Cons of int * list
let rec scs l =
  match l with
  | Nil -> Nil
  | Cons(u, us) -> Cons(1 + u, scc us)
let rec sum l =
  match l with
  | Nil -> 0
  | Cons(u, us) -> u + sum us
let rec sum_scs l = ??
let main l =
  assert (sum (scs l) = sum_scs l)
```

tup_sum_even

```
type list = Nil | Cons of int * list
let rec sum l =
  match l with
  | Nil -> 0
  | Cons(u, us) -> u + sum us
let rec even l =
  match l with
  | Nil -> Nil
  | Cons(u, us) -> Cons(u, us)
  | Cons(u1, Cons(u2, us))
    -> Cons(u1, even us)
let rec tup_sum_even l = ??
let main l =
  assert ((sum l, even l) = tup_sum_even l)
```

pred_dup

```
type list = Nil | Cons of int * list
let rec succ l =
  match l with
  | Nil -> Nil
  | Cons(u, us) -> Cons(u + 1, succ us)
let rec dup l =
  match l with
  | Nil -> Nil
  | Cons(u, us) -> Cons(u, Cons(u, dup us))
let rec pred_dup l = ??
let main l =
  assert (succ (pred_dup l) = dup l)
```