顕在的契約計算における代数的データ型

関山 太朗 [1], 西田 雄気 [2], 五十嵐 淳 [1]

[1] 京都大学大学院情報学研究科　[2] 京都大学工学部情報学科

{t-sekiym,nishida,igarashi}@fos.kuis.kyoto-u.ac.jp

概 要　顕在的契約計算は単純な型では捉えられないプログラムの性質—例えば 0 での除算ができないといった制約—を表わしたソフトウェア契約を型システムに取り入れた型付計算体系である．契約は篩 (ふるい) 型 (refinement type) として静的型情報に現われるが，成り立つことが静的に保証できない契約はキャストにより実行時検査される．

本研究では代数的データ型を導入した顕在的契約計算を与え，その性質について議論する．本研究で扱う代数的データ型は，構築子への契約の付与や項変数の抽象化により，リストの昇順性のようなデータ構造要素間の不変条件を型として表現できるだけでなく，構築子の対応関係を明示した代数的データ型間のキャストにより不変条件の成立の可否を実行時検査できる．また本論文ではデータ構造に関する実行時検査の遅延化や計算体系の実装についても論ずる．

1　はじめに

ソフトウェア契約は整数型などの単純な型では捉えられないプログラムの性質—例えば割り算では 0 除算ができないといった制約—を表したもので，プログラマがプログラム片に対して要求する仕様として与えられる．契約機構は様々なプログラミング言語で利用することができ，例えば C 言語では assert による不変式，Eiffel [15] などのオブジェクト指向プログラミング言語ではメソッド呼び出しの前後で検査される事後条件・事前条件・不変条件として契約を記述することができる．関数型プログラミング言語については，Findler らが高階関数に対する契約機構を提案し，その仕組みを内蔵した計算体系として (潜在的) 契約計算を与えた [7]．Findler らが与えた計算体系では，Eiffel などと同様，契約は全て実行時に検査される．

契約を静的に検査する計算体系としては Flanagan らが与えた顕在的契約計算 [13] がある．この計算体系は契約を篩 (ふるい) 型 (refinement type) として静的型システムに取り入れることで，契約情報を型に関する様々な構造—例えば代数的データ型—と組み合わせたり，契約検査に型を用いた静的解析を適用することを容易にしている．篩型の例として，0 より大きな整数を表わす篩型は $\{x{:}\mathsf{int} \mid 0 < x\}$ と書け，真偽値式 $0 < x$ が整数型 int に関する契約に当たる．しかし，与えられたプログラム片が契約を満たすことを静的に判定することが一般には困難であるのと同様に，与えられたプログラム片に篩型が付くかを静的に判定することは難しい．そこで Flanagan らは静的に成立が保証できない契約についてはプログラムの実行時に検査する機構を与えた．顕在的契約計算における実行時検査は型変換 (キャスト) によって実現されており，関数型間のキャストについては Findler らのアイディアが用いられている．しかし顕在的契約計算でデータ構造とその型を扱うための研究は，我々の知る限り，これまでにあまりされてこなかった．

本研究では Belo らが提案した枠組み [1] に基づいて代数的データ型を備えた顕在的契約計算 λ^H_μ を与え，その性質について議論する．本論文で扱う代数的データ型と λ^H_μ の特徴は次の通りである．

- データ構築子への契約の付与や項変数の抽象化により，リスト要素が昇順で並んでいるといったようなデータ構造要素間の不変条件を代数的データ型として表現できる．

- 異なる代数的データ型間のキャストによる型変換をデータ構築子の対応関係を明示することで実現し，それによりデータ構造の不変条件の成立の可否を実行時検査できる．例えば整数リストとその昇順リストの間には構築子間の対応関係を与えることができ，整数リストから昇順リストへ，またはその逆の実行時型変換が可能である．

また本研究の貢献は次の通りである．

- 上記の特徴をもった計算体系の形式化と型健全性の証明．
- 実行時検査の実行を遅延させる意味論の形式化と，正格な実行時検査との対応の証明．
- プリプロセッサ Camlp4 を用いたプログラミング言語 OCaml 上の λ_μ^H の実装．

本論文は次のように構成される．第 2 節では代数的データ型によるデータ構造上の契約表現や異なる代数的データ型間でキャストを行う方法について述べる．第 3 節では顕在的契約計算 λ_μ^H の形式的定義を与え，その型健全性を示す．また第 4 節では代数的データ型に関する実行時検査の遅延化，第 5 節では λ_μ^H の実装について論ずる．そして第 6 節で本研究と先行研究について比較した後，第 7 節で本論文のまとめと将来の課題について述べる．

2 顕在的契約計算への代数的データ型の導入

本節では整数リストを例に，顕在的契約計算でのデータ構造に関する契約情報の表現方法について述べる．そのためにまず顕在的契約計算について簡単に振り返り，その後データ構造に関する契約を表現するためには素朴に篩型を使用するだけでは不十分であることを見る．次に，データ構築子への契約の付与や項変数の抽象化を許す代数的データ型を用いることで，代数的データ型がデータ構造上の様々な契約を効果的に表現できることを示す．また，データ構築子の対応関係を明示することで，代数的データ型に対して実行時検査を行う方法についても述べる．最後に，契約の代数的データ型による表現と篩型による表現を比較する．

2.1 顕在的契約計算概略

顕在的契約計算は契約の静的検査と実行時検査のための機構を備えた計算体系で，成立が静的に証明できない契約についてはプログラム実行時に成り立つかどうかを検査する．契約の静的検査は型システムに取り込まれており，契約情報は篩型を用いて記述される．篩型 $\{x{:}T \mid e\}$ は変数 x，型 T，真偽値型の項 e から構成される型で，型 T の値 v のうち，$e\{v/x\}$ が true に評価されるようなものの集合を表わす．例えば整数型 int を用いると，正整数を表わす型は $\{x{:}\mathsf{int} \mid 0 < x\}$ と書ける．

契約の実行時検査は型変換 (キャスト) によって実現される．キャスト $\langle T_1 \Leftarrow T_2 \rangle^\ell$ は型 T_2 の値に適用されると，その値が型 T_1 として振る舞うことができるかを検査する．検査に成功すると値が返り，失敗すると例外 $\Uparrow\ell$ が発生する．ここで ℓ はキャストごとに付与されるラベルで，各キャストにはそれぞれ異なるラベルが与えられる．例えば整数型から正整数型へのキャストは $\langle \{x{:}\mathsf{int} \mid 0 < x\} \Leftarrow \mathsf{int} \rangle^\ell$ と書ける．整数 5 は正整数であるため，キャストによる実行時検査は成功し，$\langle \{x{:}\mathsf{int} \mid 0 < x\} \Leftarrow \mathsf{int} \rangle^\ell 5$ の値は 5 となる．一方整数 0 は正ではないため正整数型へのキャストは失敗し式 $\langle \{x{:}\mathsf{int} \mid 0 < x\} \Leftarrow \mathsf{int} \rangle^\ell 0$ を評価すると，例外 $\Uparrow\ell$ が発生する．また，依存関数型や篩型間のキャストを行うことも可能である．

2.2 データ構造に対する篩型

本研究では顕在的契約計算にリストなどの再帰的データ構造を導入するが，データ構造に関する契約を素朴に表現した篩型はデータ構造を取る述語関数によって決められる．例えばリスト要素が昇順に並んでいることを表わす篩型は，リスト要素の昇順性を判定する関数 sorted と整数リスト

型 int list を用いると，$\{x$:int list$\mid$sorted $x\}$ と書ける．以下では $\{x$:int list$\mid$sorted $x\}$ を sorted_list と書く．

しかしデータ構造の契約を表現するために素朴に篩型を使用すると，契約の静的検査と実行時検査のどちらについても問題が起こる．この問題を考察するために，まずプログラム例として挿入ソート関数 insert_sort について考える．関数 insert_sort は補助関数 insert を用いて定義され，これらは ML 風の構文を用いると次のように記述できる．

```
let rec insert (x:int) (l:sorted_list) : sorted_list = match l with
  | []     -> ⟨sorted_list ⇐ int list⟩^ℓ [x]
  | y::ys -> if x < y then ⟨sorted_list ⇐ int list⟩^ℓ (x::l)
             else ⟨sorted_list ⇐ int list⟩^ℓ
                    (y::(insert x (⟨sorted_list ⇐ int list⟩^ℓ ys)))

let rec insert_sort (l:int list) : sorted_list = match l with
  | []     -> []
  | x::xs -> insert x (insert_sort xs)
```

これらの関数は灰色部分のキャストを除けば通常の挿入ソート関数を定義している．関数 insert 中で期待する型と実際の型が異なる部分式は [x]，x::l，y::(insert x ys)，ys の四つで [1]，それぞれ期待する型は sorted_list であるのに対し実際の型は int list である．もし四つの部分式それぞれについて，関数 sorted を適用した際 true を返すことが静的に証明できればキャストは必要ない．そうでなく，もしある部分式が契約を満たすかどうかわからない場合は，静的型情報を強制的に書き換えるためにキャストを挿入することになる．

篩型を用いた静的契約検査の際に起こる問題として，篩型が表わすデータ構造の構成要素の性質や構成要素間の関係といった契約情報が構成要素自体の型に現われないという点が挙げられる．例えば関数 insert では，l の型は sorted_list であるが，その部分リストである ys はリストのデータ構築子::から得られたものであるため，その型は単に int list となる．そのため insert の再帰呼び出しの部分で ys に sorted_list が付くことを証明する際には何らかの静的解析手法を用いる必要がある．

実行時検査に関する問題としては，不要な契約検査が発生することが考えられる．ここで仮に関数 insert の部分式 x::l が sorted の表わす契約を満たすことを静的に証明できなかったとしよう．このとき int list から sorted_list へのキャストを x::l に適用することになり，実行時には例えば (1) x の表わす値が l の表わすリストの先頭要素より小さいことと，(2) そのリスト自体も昇順に並べられていることが検査される．しかし変数 l の型は sorted_list であることから (2) の条件は明らかに成り立ち，その実行時検査は常に成功することがわかる．

2.3 代数的データ型による契約表現

データ構造上の契約を記述するいくつかの研究ではそのデータ構築子に契約 (述語) を付与する方法をとっている [11, ?, 14, 18, 8, 12, 3, 4, 16] が，これは上記の問題を解決するためにも適用できる．データ構築子に契約が付与できるようになると，リストが昇順に並べられていることは次のような代数的データ型 sorted_list′ によって表現できる．

```
type sorted_list' =
  SNil  of unit
| SCons of x:int × {xs:sorted_list'|nil xs || x < head xs}
```

ここで x:T × T' は依存積型 (dependent product type) で，型 T' は型 T の変数 x を用いて記述される．また関数 nil は引数のリストが空であるかを返し，関数 head は引数リストの先頭要素を返す関数である．従って sorted_list′ のデータ構築子 SCons は，その引数が整数と型 sorted_list′ のリ

[1] 型 sorted_list の式は明らかに型 int list の式と見做せる．

ストで，さらに昇順リスト xs が空であるか，そうでなければ xs の先頭要素は整数 x よりも大きいことを要求していることになる．

この代数的データ型を用いるとデータ構造の構成要素の型に契約情報が現われ，関数 insert の例では昇順リストの部分リストを表わす変数 ys の型は sorted_list' となる [2]．また不要な実行時検査も軽減でき，例えば関数 insert で SCons を用いて先頭要素が x で残りの部分リストが l になるような昇順リストを構築しようとし，x と l が SCons の引数型が表わす契約を満たすかを実行時検査する際には，l の型は sorted_list′ であるため，前述の条件 (1) だけが検査される．

また，契約を表わしたプログラム関数がデータ構造以外の補助引数を取るような場合に対応するために代数的データ型の項変数による抽象化 [14, 12, 16] を導入する．例えばリストには与えられた整数 n より大きい要素しかないことを表わす代数的データ型は次のように定義できる．

```
type greater_list (n:int) =
  GNil  of unit
| GCons of {x:int|n < x} × greater_list n
```

さらに代数的データ型の項変数の抽象化に伴ない，代数的データ型に項を適用することも許す．すなわち整数 0 より大きいものしか含まないようなリストの型は greater_list 0 と書ける．

2.4 代数的データ型変換

契約情報を取り入れた代数的データ型は，汎用なデータ構造のうち構成要素についてある関係や性質が成り立つものの集合とみなすことができるため，篩型と同様に，契約情報を含む代数的データ型に属するデータ構造は元の汎用なデータ構造としても扱え，さらにその逆も可能であることが望ましい．しかし既存先行研究ではデータ構造を異なる代数的データ型へ変換するには，プログラマが明示的に変換関数を記述する必要があった．

本研究の計算体系では，異なる代数的データ型の間であってもデータ構築子間の対応さえ明示すれば自動的なデータ構造の変換を可能にする．代数的データ型 sorted_list′ の例であれば，リストのデータ構築子には空リストを表わす Nil と先頭要素と残りの部分リストから新しいリストを構築する Cons があるとすると，次のように SNil が Nil に，SCons が Cons に対応することを明示することで，sorted_list′ から int list へ，もしくは int list から sorted_list′ へのキャストが利用できる．

```
type sorted_list' =
  SNil  || Nil  of Unit
| SCons || Cons of x:int × {xs:sorted_list'| nil xs || x < head xs }
```

このときキャスト $\langle \text{sorted\_list}' \Leftarrow \text{int list} \rangle^{\ell}$ の適用結果は，篩型の場合と同様に，実行時検査が成功すれば sorted_list′ の値に，失敗すれば例外になる．

$$\langle \text{sorted\_list}' \Leftarrow \text{int list} \rangle^{\ell}\,(1 :: 2 :: []) \longrightarrow^{*} \text{SCons}\,(1, (\text{SCons}\,(2, \text{SNil})))$$
$$\langle \text{sorted\_list}' \Leftarrow \text{int list} \rangle^{\ell}\,(2 :: 1 :: []) \longrightarrow^{*} \Uparrow\ell$$

この構築子間の対応は多対多となることも許しており，そうすることで初めて元の汎用なデータ構造と互換となる代数的データ型も存在する．その例として，与えられた整数 n を少なくとも一つ含むという整数リストの契約を表現した代数的データ型 list_including を考えてみる．

```
type list_including (n:int) =
  LConsEq  || Cons of {x:int|x = n} × int list
| LConsNEq || Cons of {x:int|x <> n} × list_including n
```

まず空リストの場合は上記の契約を満たさないため，list_including は Nil に対応するデータ構築子を含まない．また Cons に対応するデータ構築子は二つ与えており，一つは先頭の整数が n と等しい場合，もう一方は先頭は n と等しくなくかつ残りの部分リストに n と等しい整数が存在する場合

[2] 正確には {xs:sorted_list'|nil xs || y < head xs} となる．

である．このような代数的データ型を与えた場合，多対多のデータ構築子間の対応を許すことで自然数リストとの相互変換が可能となる．型 int list から型 list_including へのキャストの振る舞いとしては，空リストに適用されると例外が発生し，非空リストに適用されると Cons に対応するデータ構築子として LConsEq と LConsNEq のうちどちらか一方を選ぶことになる．

$$\begin{array}{lll} \langle \mathsf{list\_including}\, 0 \Leftarrow \mathsf{int\ list} \rangle^{\ell}\, [\,] & \longrightarrow^{*} & \Uparrow \ell \\ \langle \mathsf{list\_including}\, 0 \Leftarrow \mathsf{int\ list} \rangle^{\ell}\, (2 :: 0 :: [\,]) & \longrightarrow^{*} & \mathsf{LConsNEq}\, (2, (\mathsf{LConsEq}\, (0, \mathsf{Nil}))) \end{array}$$

ただし LConsEq と LConsNEq のように変換候補として複数のデータ構築子が存在する場合，本研究ではその選び方までは規定せず，決定的な選択方法であればどのように選んでもよいとしている．

2.5 代数的データ型による契約表現: 利点と欠点

利点: 契約を代数的データ型によって表現することの最大の利点は実行時契約検査を効率的に行うことができる点である．これは契約に対応する代数的データ型を用いて構成されたデータ構造には，それがどのような実行時パスを通ることで契約検査に成功したかが記録されているためである．例えば前述のある整数 n を少なくとも一つ含む整数リストという契約について考えてみよう．この契約を篩型で表現した場合，その篩型が付くリストは確かにある整数 n を持つリストであることが保証されるが，それがどの位置にあるかまでは示されていない．そのためリストの要素のうち n と等しい要素を得るには実行時検査による確認が必要となる．一方代数的データ型による表現ではデータ構築子に先頭要素が n と等しいかどうかという情報が含まれているため，実行時検査を行うことなく，リストの要素がある整数 n と等しいことを示す「証拠」を得ることができる．また第 2.3 節で示したように，篩型では不要な実行時検査が起こるが代数的データ型では起こらない場合もある．

代数的データ型による契約表現には契約の静的検査でも利点をもつ場合がある．これは特にいくつかの先行研究 [18, 12, 20, 16] にあるように，契約検査を静的にのみ行うような場合に顕著であると考えられる．このような場合には実行時検査によって契約が成り立つことの証拠を得ることができないため，データ構築子への契約の付与が静的契約検査に大きく作用する．これらの研究から見た本研究の位置付けは対象となる計算体系の形式化である [16]．ただし現段階では本研究における計算体系は静的契約検査には言及しておらず，実際に我々の計算体系を基礎体系として利用するためには静的契約検査に関する研究を進める必要がある．一方契約検査を静的かつ動的に行う場合には上記で述べたように実行時検査によって契約成立の証拠を得ることができるため，現段階では静的検査という観点からは先行研究 [19] と比べそれほど大きな利点はないと思われる．

欠点: 契約を代数的データ型によって表現することの欠点として，契約が成り立つようにデータ構造を構築しなくてはならないことが挙げられる．例えば `list_including` については，リストのデータ構築子 `Cons` に対応するデータ構築子が二つ存在するため，`list_including` のリストを構築する際には先頭要素がある整数 n であるかどうかで二つの構築子を使い分けなくてはならない．一方篩型ではそのような使い分けは必要はなく，先頭要素が n であろうがなかろうが統一的に Cons を用いればよい．

また代数的データ型によって表現された契約は，篩型によって表現されたものよりも，合成などの操作を適用しにくい．例えば第 2.2 節で insert_sort の戻り値が昇順に並べられていることに加え，引数リストの置換になっていることを保証したいとする．このとき本論文の計算体系では，昇順であることと置換であることの両方を表わす新たな代数的データ型を定義するか，単に置換であることを代数的データ型 sorted_list′ 上の契約として与える (もしくはその逆) しかなく，二つの契約がどちらも代数的データ型として表現されていたとしても，それらから新しい代数的データ型を与える手段はまだない．一方篩型を用いた場合は単に二つの契約の論理積をとればよい．

$$
\begin{array}{llll}
\textbf{型} & \mathsf{B} & ::= & \mathsf{unit} \mid \mathsf{Bool} \\
& T & ::= & \mathsf{B} \mid x{:}T_1 \to T_2 \mid \{x{:}T \mid e\} \mid \\
& & & x{:}T_1 \times T_2 \mid \tau\,\{e\} \\
\\
\textbf{項} & c & ::= & () \mid \mathsf{true} \mid \mathsf{false} \\
& e & ::= & c \mid x \mid \texttt{<<no parses (char 13): \f(x:T1):T2.e*** >>} \mid \langle T_1 \Leftarrow T_2 \rangle^{\ell} \mid e_1\, e_2 \mid (e_1, e_2) \mid e.1 \mid e.2 \mid C\{e_1\}e_2 \mid \\
& & & \mathsf{match}\ e\ \mathsf{with}\ \overline{C_i\, x_i \to e_i}^{\,i} \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3
\end{array}
$$

$$
\begin{array}{llll}
\textbf{型環境} & \Gamma & ::= & \varnothing \mid \Gamma, x{:}T \\
& \varsigma & ::= & \mathsf{type}\,\tau\,(x{:}T) = \overline{C_i : T_i}^{\,i} \mid \\
& & & \mathsf{type}\,\tau\,(x{:}T) = \overline{C_i \parallel D_i : T_i}^{\,i} \\
& \Sigma & ::= & \varnothing \mid \Sigma, \varsigma
\end{array}
$$

図 1. 構文.

3 顕在的契約計算 λ_{μ}^{H}

本節では第 2 節で述べたような代数的データ型を備えた顕在的契約計算 λ_{μ}^{H} の形式的定義を与え，その型健全性を示す．計算体系 λ_{μ}^{H} の定義は Belo らの手法 [1] と同様に行う．Knowles らが契約の静的検査を計算体系の一部として与えた [13] のに対し，Belo らは計算体系の定義やそのメタ理論を簡潔にするために静的契約検査機構を計算体系自体には含めず，必ず成り立つような実行時契約検査をプログラム実行前に除去することで契約の静的検査を実現している．本論文でも Belo らと同様に計算体系の一部としては静的契約検査機構は与えない．また必ず成り立つような実行時検査の定義とその除去がプログラムの実行に影響しないことの証明は将来の課題とし，本論文では契約の実行時検査に焦点を当てる．

いくつかの先行研究 [12, 13, 16] の型付け規則では，変数や定数に付く型はより正確な意味を捉えたものであったり，if 式や match 式などの条件分岐式では，どのような式によりどう分岐したかまでが追跡できるようになっている．そのような型付け規則を λ_{μ}^{H} に対して与えることもできるが，型システムの定義が複雑になり，また本論文の趣旨は静的検証そのものではないため，本論文では Belo らに倣いより単純な型付け規則を与える．ただし，先行研究のようなよりきめ細かな型付け規則を λ_{μ}^{H} に対し与えることは容易で，本論文で述べる性質には影響しないと考えている．

計算体系の説明では主に Belo らとの違いについて述べる．またいくつかの関係やメタ関数などは直観的な意味を記述するに留め，形式的な定義は与えないものとする．以下では列を上付き横棒によって記述する．例えばデータ構築子の列は添字 i を用いて $\overline{C}^{\,i}$ と記す．

3.1 構文

顕在的契約計算体系 λ_{μ}^{H} のプログラム構文を図 1 に示す．ここでは例外などの，プログラムの実行中に登場するような項は含まれていない．それぞれ基本型をメタ変数 B，型を T，定数を c，データ構築子を C または D，項を e，型環境を Γ，代数的データ型定義を ς，型定義環境を Σ によって表わす．型は unit と Bool を含む基本型，依存関数型，篩型に加え依存積型と代数的データ型から成る．代数的データ型名は τ と書き，項 e への τ の適用を $\tau\,\{e\}$ と書く．

プログラムは定数，変数，再帰関数，キャスト，関数適用，組，射影，データ構築子の適用，match 式，if 式から成る．再帰関数とデータ構築子の適用以外の要素は先行研究と同様か，もしくは標準的なものであるためここでは説明を省略する．再帰関数 $\mathtt{fix}\, f(x{:}T){=}\, e$ は e 中の変数 x と f を束縛しており，x は再帰関数に渡された値を，f は再帰関数自身を表わす変数である．項 $C\{e_1\}e_2$ はデータ構築子 C を型への引数 e_1 と構築子自身の引数 e_2 に適用していることを意味する．

型環境 Γ は変数と型の束縛列として与えられ，型環境が束縛する変数名はそれぞれ異なるとする．また代数的データ型を定義する構文は二種類与えられている．型定義 $\mathsf{type}\,\tau\,(x{:}T) = \overline{C_i : T_i}^{\,i}$ は引数 x として型 T の値をとる代数的データ型 τ の定義を表わす．データ構築子定義列 $\overline{C_i : T_i}^{\,i}$ は τ のデータ構築子 C_i の引数型が T_i であることを宣言している．型定義 $\mathsf{type}\,\tau\,(x{:}T) = \overline{C_i \parallel D_i : T_i}^{\,i}$

$\boxed{e_1 \rightsquigarrow e_2}$ **簡約規則**

<<no parses (char 15): (\f(x:T1):T2.e)*** v >> $\rightsquigarrow$ <<no parses (char 20): e{v/x,\f(x:T1):T2.e/***f} >>

$(v_1, v_2).1 \rightsquigarrow v_1$ R_Proj1

$\mathsf{if\,true\,then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow e_1$

$(v_1, v_2).2 \rightsquigarrow v_2$ R_Proj2

$\mathsf{if\,false\,then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow e_2$

$\mathsf{match}\ C_j\{e\}v\ \mathsf{with}\ \overline{C_i\ x_i \to e_i}^{\,i} \rightsquigarrow e_j\{v/x_j\}$ (ただし $C_j \in \overline{C_i}^{\,i}$ の場合

$\langle \mathsf{B} \Leftarrow \mathsf{B} \rangle^\ell\, v \rightsquigarrow v$

$\langle x{:}T_{11} \to T_{12} \Leftarrow x{:}T_{21} \to T_{22} \rangle^\ell\, v \rightsquigarrow (\lambda\, x{:}T_{11}.\mathsf{let}\ y = \langle T_{21} \Leftarrow T_{11} \rangle^\ell\, x\ \mathsf{in}\ \langle T_{12} \Leftarrow T_{22}\{y/x\} \rangle^\ell\, (v\ y))$ (ただし y は新しい変

$\langle x{:}T_{11} \times T_{12} \Leftarrow x{:}T_{21} \times T_{22} \rangle^\ell\, (v_1, v_2) \rightsquigarrow \mathsf{let}\ y = \langle T_{11} \Leftarrow T_{21} \rangle^\ell\, v_1\ \mathsf{in}\ (y, \langle T_{12}\{y/x\} \Leftarrow T_{22}\{v_1/x\} \rangle^\ell\, v_2)$ (ただし y は新しい変

$\langle T_1 \Leftarrow T_2 \rangle^\ell\, v \rightsquigarrow \mathit{check\_refines}(T_1, \langle \mathit{unref}(T_1) \Leftarrow \mathit{unref}(T_2) \rangle^\ell\, v)^\ell$ (ただし T_1 もしくは T_2

$\mathsf{check}\,(\{x{:}T \mid e\}, v)^\ell \rightsquigarrow \langle \{x{:}T \mid e\}, e\{v/x\}, v \rangle^\ell$

$\langle \{x{:}T \mid e\}, \mathsf{true}, v \rangle^\ell \rightsquigarrow v$ R_OK

$\langle \{x{:}T \mid e\}, \mathsf{false}, v \rangle^\ell \rightsquigarrow \Uparrow \ell$

$\boxed{\Sigma \vdash e_1 \twoheadrightarrow e_2}$ **代数的データ型変換スキーム**

$$\frac{C_1 \in \mathit{CompatCtrsOf}_\Sigma(\tau_1, C_2) \quad \overline{\mathit{ArgTypeOf}_\Sigma(\tau_i) = x_i{:}T_i}^{\,i\in\{1,2\}} \quad \overline{\mathit{CtrArgOf}_\Sigma(C_i) = T_i'}^{\,i\in\{1,2\}}}{\Sigma \vdash \langle \tau_1\{e_1\} \Leftarrow \tau_2\{e_2\} \rangle^\ell\, C_2\{e\}v \twtwoheadrightarrow C_1\{e_1\}(\langle T_1'\{e_1/x_1\} \Leftarrow T_2'\{e_2/x_2\} \rangle^\ell\, v)}\ \text{RC\_Datatype}$$

$$\frac{\mathit{CompatCtrsOf}_\Sigma(\tau_1, C_2) = \varnothing}{\Sigma \vdash \langle \tau_1\{e_1\} \Leftarrow \tau_2\{e_2\} \rangle^\ell\, C_2\{e\}v \twoheadrightarrow \Uparrow \ell}\ \text{RC\_Blame}$$

$\boxed{\langle \Sigma, \to \rangle \vdash e_1 \longrightarrow e_2}$ **評価規則**

$$\frac{e_1 \rightsquigarrow e_2}{\langle \Sigma, \to \rangle \vdash E[e_1] \longrightarrow E[e_2]}\ \text{E\_Red} \qquad \frac{\Sigma \vdash e_1 \to e_2}{\langle \Sigma, \to \rangle \vdash E[e_1] \longrightarrow E[e_2]}\ \text{E\_Eager} \qquad \frac{E \neq [\,]}{\langle \Sigma, \to \rangle \vdash E[\Uparrow \ell] \longrightarrow \Uparrow \ell}\ \text{E\_Blame}$$

図 2. 意味論.

は，さらに τ のデータ構築子 C_i がデータ構築子 D_i と互換であることを宣言している．型定義環境 Σ は代数的データ型の定義列で，型定義環境中の代数的データ型やそのデータ構築子の名前は一意に定まるとする．

本論文では構文についての標準的な記法をいくつか導入する．項 e に自由に出現する変数の集合を $\mathsf{FV}(e)$ とし，項 e' を e 上の自由変数 x に capture avoiding に代入して得られる項を $e\{e'/x\}$ と書く．さらに自由変数が存在しない項を閉じた項と呼び，α 同値な項は同じ項とみなす．型や型環境，さらに後ほど与える実行時項についても同様である．

また関数と関数適用に関する糖衣構文を与える．依存関数型 $x{:}T_1 \to T_2$ について x が T_2 に出現しないとき，単に $T_1 \to T_2$ と書く．また再帰関数 $\mathtt{fix}\ f(x{:}T) = e$ において f が e に自由に出現しないような場合については $\lambda\, x{:}T.e$ と書く．さらに let 式 $\mathsf{let}\, x = e_1\ \mathsf{in}\ e_2$ は適切な型 T を用いた関数適用 $(\lambda\, x{:}T.e_2)\, e_1$ の糖衣構文とする．

3.2 操作的意味論

本節では計算体系 λ_μ^H に値呼び評価戦略による意味論を与える．そのためにまず例外や契約検査の最中であることを表わす項などを構文に加える．次に，意味論を三つの関係，すなわち簡約関係

$\rightsquigarrow$，代数的データ型変換スキーム (以下単にデータ型変換スキーム) $\twoheadrightarrow$，評価関係 $\longrightarrow$ として与える．簡約関係と評価関係は項の簡約と評価を表わした関係で，データ型変換スキームは代数的データ型間キャストを適用した際の振る舞いについての条件を規定する関係である．簡約関係は閉じた項上の二項関係で，データ型変換スキームは代数的データ型の情報を必要とするため型定義環境と閉じた二つの項上の関係として定義する．評価関係は項の評価が決定的になるようにデータ型変換スキームの部分関係について抽象化されており，型定義環境，データ型変換スキームの部分関係，そして閉じた二つの項上の関係として与える．図 2 はこれらの導出規則を示している．本論文では関係 R が三つ組 (Σ, e_1, e_2) (または (Σ, T_1, T_2)) を含むことを $\Sigma \vdash e_1 R e_2$ (または $\Sigma \vdash T_1 R T_2$) と書く．また評価関係の反射的推移的閉包を $\longrightarrow^*$ と書く．

計算体系 λ_μ^H の意味論を与えるために，まず λ_μ^H の値と実行時項を定義する．実行時状態として契約検査と例外 $\Uparrow \ell$ を加えた項と値の構文は次のようになる．

$$
\begin{array}{rcl}
v & ::= & c \mid \mathtt{fix}\ f(x{:}T) = e \mid \langle T_1 \Leftarrow T_2 \rangle^\ell \mid (v_1, v_2) \mid C\{e\}v \\
e & ::= & \ldots \mid \langle \{x{:}T \mid e_1\}, e_2, v \rangle^\ell \mid \mathsf{check}\,(\{x{:}T \mid e_1\}, e_2)^\ell \mid \Uparrow \ell
\end{array}
$$

契約検査のとる形には待機中検査と活性検査がある．待機中検査 $\mathsf{check}\,(\{x{:}T \mid e_1\}, e_2)^\ell$ は e_2 の評価結果が契約 e_1 を満たすかを検査することを意味している．活性検査 $\langle \{x{:}T \mid e_1\}, e_2, v \rangle^\ell$ は v が契約 e_1 を満たすかを検査している最中の状態を表わしており，e_2 は $e_1\{v/x\}$ の評価途中の項である．活性検査は先行研究にも存在した [13, 1] が，待機中検査は第 4 節で二つの意味論を比較しやすくするために本研究で新たに導入したものである．

簡約関係 $\rightsquigarrow$ の導出規則はキャストと契約検査に関するもの以外は標準的であるためここでは説明を省略する．キャストに関する規則は (代数的データ型を除く) 型の各種類についてそれぞれ定義されている．簡約規則 R_Base は基本型間のキャストのための規則で，引数の値をそのまま返す．規則 R_Fun は依存関数型間のキャストの簡約規則である．先行研究 [1] と同様に簡約結果は関数であり，この関数は値 v' に適用されるとまずその値 v' に引数型間の反変的キャストを適用する．その結果にキャスト対象である v を適用し，さらにその結果に戻り値型間の共変的キャストを適用する．ただし戻り値型間のキャスト上の自由変数 x はそれぞれ適切に置き換えられる．規則 R_Prod は依存積型間のキャストの簡約規則で，そのアイディアは R_Fun と同様である．

規則 R_Refine はキャスト上の型が一方でも篩型であったときに適用される簡約規則で，まず T_1 と T_2 の構造に関するキャスト $\langle \mathit{unref}(T_1) \Leftarrow \mathit{unref}(T_2) \rangle^\ell$ を値 v に適用してから，その結果が T_1 が表わす契約を満たすかを検査する．規則 R_Refine では二つのメタレベル関数 *check_refines* と *unref* を利用しており，$\mathit{check\_refines}(T, e)^\ell$ は項 e を評価した結果が型 T 上の契約を満たすかどうかをラベル ℓ を用いて検査する項を表わし，$\mathit{unref}(T)$ は型 T の外側の契約を全て取り払った型を表わす．メタレベル関数 *check_refines* と *unref* はそれぞれ次のように定義できる．

$$
\begin{array}{lcl}
\mathit{check\_refines}(\{x{:}T \mid e_1\}, e_2)^\ell & = & \mathsf{check}\,(\{x{:}T \mid e_1\}, \mathit{check\_refines}(T, e_2)^\ell)^\ell \\
\mathit{check\_refines}(T, e)^\ell & = & e \quad \text{(ただし } T \text{ が篩型でない場合)} \\
\mathit{unref}(\{x{:}T \mid e\}) & = & \mathit{unref}(T) \\
\mathit{unref}(T) & = & T \quad \text{(ただし } T \text{ が篩型でない場合)}
\end{array}
$$

規則 R_Check は待機中検査を活性検査に簡約するために適用される．活性検査は契約の検査結果が true になれば検査対象の値を返し (規則 R_OK)，false になれば契約違反として例外 $\Uparrow \ell$ を発生させる (規則 R_Fail)．

データ型変換スキーム $\twoheadrightarrow$ は二つの規則から導くことができ，どちらも代数的データ型間のキャスト $\langle \tau_1\{e_1\} \Leftarrow \tau_2\{e_2\} \rangle^\ell$ を $C_2\{e\}v$ によって表現されたデータ構造に適用した際に使用される．変換規則 RC_Datatype は代数的データ型 τ_1 のデータ構築子から C_2 と互換なもの (規則中の C_1 に相当する) を適当に選び，C_2 の引数 v を選んだデータ構築子の引数型にキャストする．ただし C_1，C_2 の引数型はそれぞれ代数的データ型の引数 x_1，x_2 に依存しているため，キャストに与えられた項 e_1，e_2 を代入する．この変換を実現するために RC_Datatype は型定義に関する情

報を取り出す三つのメタレベル関数を使用する．メタレベル関数 $\mathit{CompatCtrsOf}$ は型定義環境 Σ，代数的データ型 τ，データ構築子 C を受け取ると，Σ 中の τ のデータ構築子のうち C と互換であるようなものの集合を返す部分関数である．例えば第 2 節で与えた list_including が型定義環境 Σ に含まれているとすると，$\mathit{CompatCtrsOf}_\Sigma(\mathsf{list\_including}, \mathsf{Cons}) = \{\mathsf{LConsEq}, \mathsf{LConsNEq}\}$ であり $\mathit{CompatCtrsOf}_\Sigma(\mathsf{list\_including}, \mathsf{Nil}) = \{\}$ となる．また $\mathit{ArgTypeOf}_\Sigma(\tau)$ は Σ 中における τ が取る引数とその型を返す．さらにメタレベル関数 $\mathit{CtrArgOf}$ は型定義環境 Σ とデータ構築子 C を受け取ると Σ 中で与えられている C の引数型を返す．これらを用いると RC_DATATYPE は図 2 のように書ける．もう一方の変換規則 RC_BLAME は τ_1 のデータ構築子に C_2 と互換なものがなかった場合に適用される規則で，この場合は適切なデータ構築子が存在しないため例外 $\Uparrow \ell$ が変換結果となる．

ここで，評価関係がデータ型変換スキームを直接用いると項の評価が非決定的になることに注意されたい．これは規則 RC_DATATYPE ではキャスト後のデータ構築子 (C_1 に相当するもの) を一つに決定しておらず，単に C_2 と互換なデータ構築子集合 $\mathit{CompatCtrsOf}_\Sigma(\tau, C_2)$ の中から適当に選ぶということしか述べていないためである．特に C_2 と互換なデータ構築子が複数存在する場合，データ型変換スキームによる項 $C_2\{e\}v$ の変換が常に同じ結果になるとは限らない．

計算体系 λ_μ^H の意味論にこのような非決定性を許すと λ_μ^H に関するいくつかの性質が成り立たなくなる，もしくはその証明が困難になるため，評価関係の定義には変換が決定的になるようデータ型変換スキームの部分関係である代数的データ型変換関係を用いる．以下では代数的データ型変換関係を単にデータ型変換関係と呼び，メタ変数 $\twoheadrightarrow$ を使って表わす．

定義 1 (データ型変換関係)**.** データ型変換スキームの部分関係 $\twoheadrightarrow$ について次のことが成り立つとき，$\twoheadrightarrow$ をデータ型変換関係と呼ぶ．

1. 任意の Σ, e_1, e_2 について，$\Sigma \vdash e_1 \twoheadrightarrow e_2$ ならば，ある項 e_2' が存在し $\Sigma \vdash e_1 \twoheadrightarrow e_2'$.
2. 任意の Σ, e_1, e_2, e_2' について，$\Sigma \vdash e_1 \twoheadrightarrow e_2$ かつ $\Sigma \vdash e_1 \twoheadrightarrow e_2'$ ならば，$e_2 = e_2'$.

項の評価は評価関係 $\longrightarrow$ を用いて $\langle \Sigma, \twoheadrightarrow \rangle \vdash e_1 \longrightarrow e_2$ と書き，これは与えられた型定義環境 Σ とデータ型変換関係 $\twoheadrightarrow$ の下で，項 e_1 は 1 ステップで e_2 へ簡約されることを意味している．評価関係の導出規則は簡約基を決定するために評価文脈 [6] を用いて定義される．メタ変数 E で表わす λ_μ^H の評価文脈は次のように与えられる．

$$
\begin{array}{rcl}
E & ::= & [\,] \mid E\ e_2 \mid v_1\ E \mid (E, e_2) \mid (v_1, E) \mid E.1 \mid E.2 \mid C\{e_1\}E \mid \mathsf{match}\ E\ \mathsf{with}\ \overline{C_i\ x_i \to e_i}^{\,i} \mid \\
 & & \mathsf{if}\ E\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \langle \{x{:}T \mid e_1\}, E, v \rangle^\ell \mid \mathsf{check}\ (\{x{:}T \mid e_1\}, E)^\ell
\end{array}
$$

この評価文脈の定義により意味論の評価戦略が値呼びで最左最外簡約であることがわかる．規則 E_RED は簡約関係 $\rightsquigarrow$ による評価，E_EAGER はデータ型変換関係 $\twoheadrightarrow$ による評価を表わしており，E_BLAME は例外が発生したときはプログラムの評価結果も例外になることを意味している．

3.3 型システム

本節では λ_μ^H のプログラムと実行時項に対する型システムを与える．この型システムではプログラムと実行時項それぞれに三つの判断を与えており，代数的データ型の情報を参照するために各判断とも型定義環境をとる．実行時項の型システムは先行研究と同様に評価関係を用いるため，さらにデータ型変換関係を必要とする．プログラムの判断は，型環境と型の well-formedness 判断 $\Sigma \vdash \Gamma$ と $\Sigma; \Gamma \vdash T$，そして型付け判断 $\Sigma; \Gamma \vdash e : T$ の三つであり，それぞれの導出規則を図 3 に示している．同様に実行時項の判断も $\langle \Sigma, \twoheadrightarrow \rangle \vdash \Gamma$，$\langle \Sigma, \twoheadrightarrow \rangle; \Gamma \vdash T$，$\langle \Sigma, \twoheadrightarrow \rangle; \Gamma \vdash e : T$ の三つで，その導出規則を図 4 に示す．これらの導出規則についても多くの規則が先行研究と同様かもしくは標準的なものになっているため，ここでも特に説明が必要だと思われる規則についてだけ述べる．

$\boxed{\Sigma \vdash \Gamma}$ **型環境 Well-Formedness 規則**

$$\frac{}{\Sigma \vdash \varnothing}\ \text{WC\_Empty} \qquad \frac{\Sigma \vdash \Gamma \quad \Sigma;\Gamma \vdash T}{\Sigma \vdash \Gamma, x{:}T}\ \text{WC\_ExtendVar}$$

$\boxed{\Sigma;\Gamma \vdash T}$ **型 Well-Formedness 規則**

$$\frac{\Sigma \vdash \Gamma}{\Sigma;\Gamma \vdash \mathsf{B}}\ \text{WT\_Base} \quad \frac{\Sigma;\Gamma \vdash T_1 \quad \Sigma;\Gamma, x{:}T_1 \vdash T_2}{\Sigma;\Gamma \vdash x{:}T_1 \to T_2}\ \text{WT\_Fun} \quad \frac{\Sigma;\Gamma \vdash T_1 \quad \Sigma;\Gamma, x{:}T_1 \vdash T_2}{\Sigma;\Gamma \vdash x{:}T_1 \times T_2}\ \text{WT\_Prod}$$

$$\frac{\Sigma;\Gamma \vdash T \quad \Sigma;\Gamma, x{:}T \vdash e : \mathsf{Bool}}{\Sigma;\Gamma \vdash \{x{:}T \mid e\}}\ \text{WT\_Refine} \quad \frac{\mathit{ArgTypeOf}_\Sigma(\tau) = x{:}T \quad \Sigma;\Gamma \vdash e : T}{\Sigma;\Gamma \vdash \tau\,\{e\}}\ \text{WT\_Datatype}$$

$\boxed{\Sigma;\Gamma \vdash e : T}$ **型付け規則**

$$\frac{\Sigma \vdash \Gamma \quad c \in \mathcal{K}_\mathsf{B}}{\Sigma;\Gamma \vdash c : \mathsf{B}}\ \text{T\_Const} \quad \frac{\Sigma \vdash \Gamma \quad x{:}T \in \Gamma}{\Sigma;\Gamma \vdash x : T}\ \text{T\_Var} \quad \frac{\Sigma;\Gamma, f{:}(x{:}T_1 \to T_2), x{:}T_1 \vdash e : T_2}{\Sigma;\Gamma \vdash \mathtt{fix}\ f(x{:}T_1) = e : x{:}T_1 \to T_2}\ \text{T\_Abs}$$

$$\frac{\Sigma;\Gamma \vdash T_1 \quad \Sigma;\Gamma \vdash T_2 \quad \Sigma \vdash T_1 \parallel T_2}{\Sigma;\Gamma \vdash \langle T_1 \Leftarrow T_2 \rangle^\ell : T_2 \to T_1}\ \text{T\_Cast} \quad \frac{\Sigma;\Gamma \vdash e_1 : x{:}T_1 \to T_2 \quad \Sigma;\Gamma \vdash e_2 : T_1}{\Sigma;\Gamma \vdash e_1\, e_2 : T_2\,\{e_2/x\}}\ \text{T\_App}$$

$$\frac{\Sigma;\Gamma \vdash e_1 : T_1 \quad \Sigma;\Gamma \vdash e_2 : T_2\,\{e_1/x\} \quad \Sigma;\Gamma, x{:}T_1 \vdash T_2}{\Sigma;\Gamma \vdash (e_1, e_2) : x{:}T_1 \times T_2}\ \text{T\_Pair} \quad \frac{\Sigma;\Gamma \vdash e : x{:}T_1 \times T_2}{\Sigma;\Gamma \vdash e.1 : T_1}\ \text{T\_Proj1}$$

$$\frac{\Sigma;\Gamma \vdash e : x{:}T_1 \times T_2}{\Sigma;\Gamma \vdash e.2 : T_2\,\{e.1/x\}}\ \text{T\_Proj2} \quad \frac{\mathit{TypSpecOf}_\Sigma(C) = x{:}T_1 \twoheadrightarrow T_2 \twoheadrightarrow \tau \quad \Sigma;\Gamma \vdash e_1 : T_1 \quad \Sigma;x{:}T_1 \vdash T_2 \quad \Sigma;\Gamma \vdash e_2 : T_2\,\{e_1/x_1\} \quad \Sigma;\Gamma \vdash \tau\,\{e_1\}}{\Sigma;\Gamma \vdash C\{e_1\}\,e_2 : \tau\,\{e_1\}}\ \text{T\_Ctr}$$

$$\frac{\Sigma;\Gamma \vdash e_0 : \tau\,\{e\} \quad \mathit{CtrsOf}_\Sigma(\tau) = \overline{C_i}^{\,i\in\{1,\ldots,n\}} \quad \mathit{ArgTypeOf}_\Sigma(\tau) = y{:}T' \quad \mathit{CtrArgOf}_\Sigma(C_i) = T_i^{\,i\in\{1,\ldots,n\}} \quad \overline{\Sigma;\Gamma, x_i{:}T_i\,\{e/y\} \vdash e_i : T}^{\,i\in\{1,\ldots,n\}} \quad \Sigma;\Gamma \vdash T}{\Sigma;\Gamma \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \overline{C_i\ x_i \to e_i}^{\,i\in\{1,\ldots,n\}} : T}\ \text{T\_Match}$$

$$\frac{\Sigma;\Gamma \vdash e_1 : \mathsf{Bool} \quad \Sigma;\Gamma \vdash e_2 : T \quad \Sigma;\Gamma \vdash e_3 : T}{\Sigma;\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T}\ \text{T\_If}$$

図 3. 静的型システムの導出規則.

3.3.1 プログラムの型システム

まずプログラムの型システムについて説明する．図 3 の型環境の well-formedness 導出規則は先行研究と同様で，型の well-formedness 導出規則のうち特に重要な規則は WT_Datatype である．規則 WT_Datatype は代数的データ型の well-formedness を導出するためのもので，τ の引数型と項 e の型が一致することを確認している．

図 3 の型付け規則もその多くは先行研究と同様か標準的なもので，本研究で特に重要な規則は T_Ctr，T_Match，T_Cast である．ただし，定数の型付け規則 T_Const は基本型 B の値の集合 $\mathcal{K}_\mathsf{B}$ を使用しており，具体的には $\mathcal{K}_\mathsf{unit} = \{()\}$，$\mathcal{K}_\mathsf{Bool} = \{\mathsf{true}, \mathsf{false}\}$ と与えられる．規則 T_Ctr はデータ構築子の適用のための型付け規則である．データ構築子の適用に付く型はそのデータ構築子が属する代数的データ型であり，本研究では代数的データ型は項を引数にとるため，メタレベル関数 *TypSpecOf* を用いてデータ構築子 C が属する代数的データ型 τ とその引数の名前と型 $x{:}T_1$，そしてデータ構築子への引数の型 T_2 を型定義環境 Σ から取り出している．さらに T_Ctr は代数的データ型とデータ構築子の引数それぞれに適切な型が付くことを要求している．ただしデータ構築子の引数型 T_2 は型 T_1 の変数 x に依存していることに注意されたい．型付け規則 T_Match は match 式のための規則である．この規則は match の対象となる項 e_0 の型はある代数的データ型 τ であることを求めており，さらにパターン列 $\overline{C_i\,x_i}^{\,i\in\{1,\ldots,n\}}$ が τ の全てのデータ構築子を網羅していることを，代数的データ型のデータ構築子集合を返すメタレベル関数 *CtrsOf* を用いて表わしている．またブランチ列 $\overline{e_i}^{\,i\in\{1,\ldots,n\}}$ についてはそれぞれ適切な型環境の下で全て同じ型が付く．

$\boxed{\langle \Sigma, \rightarrow \rangle \vdash \Gamma}$ **型環境 Well-Formedness 規則** $\cdots$ $\boxed{\langle \Sigma, \rightarrow \rangle; \Gamma \vdash T}$ **型 Well-Formedness 規則** $\cdots$

$\boxed{\langle \Sigma, \rightarrow \rangle; \Gamma \vdash e : T}$ **型付け規則** $\cdots$

$$\frac{\langle \Sigma, \rightarrow \rangle \vdash \Gamma \quad \langle \Sigma, \rightarrow \rangle; \varnothing \vdash \{x{:}T \mid e_1\} \quad \langle \Sigma, \rightarrow \rangle; \varnothing \vdash e_2 : T}{\langle \Sigma, \rightarrow \rangle; \Gamma \vdash \mathsf{check}\,(\{x{:}T \mid e_1\}, e_2)^{\ell} : \{x{:}T \mid e_1\}}\ \text{T\_WaitingCheck}$$

$$\frac{\langle \Sigma, \rightarrow \rangle \vdash \Gamma \quad \langle \Sigma, \rightarrow \rangle; \varnothing \vdash \{x{:}T \mid e_1\} \quad \langle \Sigma, \rightarrow \rangle; \varnothing \vdash v : T \quad \langle \Sigma, \rightarrow \rangle; \varnothing \vdash e_2 : \mathsf{Bool} \quad \langle \Sigma, \rightarrow \rangle \vdash e_1\{v/x\} \longrightarrow^{*} e_2}{\langle \Sigma, \rightarrow \rangle; \Gamma \vdash \langle \{x{:}T \mid e_1\}, e_2, v \rangle^{\ell} : \{x{:}T \mid e_1\}}\ \text{T\_ActiveCheck}$$

$$\frac{\langle \Sigma, \rightarrow \rangle \vdash \Gamma \quad \langle \Sigma, \rightarrow \rangle; \varnothing \vdash e : T_1 \quad \langle \Sigma, \rightarrow \rangle \vdash T_1 \equiv T_2 \quad \langle \Sigma, \rightarrow \rangle; \varnothing \vdash T_2}{\langle \Sigma, \rightarrow \rangle; \Gamma \vdash e : T_2}\ \text{T\_Conv}$$

$$\frac{\langle \Sigma, \rightarrow \rangle \vdash \Gamma \quad \langle \Sigma, \rightarrow \rangle; \varnothing \vdash T}{\langle \Sigma, \rightarrow \rangle; \Gamma \vdash \Uparrow \ell : T}\ \text{T\_Blame} \qquad \frac{\langle \Sigma, \rightarrow \rangle \vdash \Gamma \quad \langle \Sigma, \rightarrow \rangle; \varnothing \vdash v : \{x{:}T \mid e\}}{\langle \Sigma, \rightarrow \rangle; \Gamma \vdash v : T}\ \text{T\_Forget}$$

$$\frac{\langle \Sigma, \rightarrow \rangle \vdash \Gamma \quad \langle \Sigma, \rightarrow \rangle; \varnothing \vdash \{x{:}T \mid e\} \quad \langle \Sigma, \rightarrow \rangle; \varnothing \vdash v : T \quad \langle \Sigma, \rightarrow \rangle \vdash e\{v/x\} \longrightarrow^{*} \mathsf{true}}{\Sigma; \Gamma \vdash v : \{x{:}T \mid e\}}\ \text{T\_Exact}$$

図 4. 実行時項型システムの導出規則.

型付け規則 T_Cast はキャストの型付け判断を導出するために適用される．キャストの型付け判断を導くためにはキャスト上の型がそれぞれ well-formed であることに加え，それらが互換であることが求められる．型間の互換性は型互換関係 $\parallel$ によって与えられ，$\Sigma \vdash T_1 \parallel T_2$ は型定義環境 Σ の下で 型 T_1 と T_2 が互換であることを意味している．紙面の都合上型互換関係の形式的な定義は与えないが，先行研究と同様に，直観的には型互換関係はキャストによる実行時検査が成功する可能性がある型同士を互換と見なす．すなわち $\Sigma \vdash T_1 \parallel T_2$ は T_1，T_2 から契約を全て除いて得られた型 T_1'，T_2' について，$T_1' = T_2'$ であるか，もしくは T_1' と T_2' が Σ の下で互換な代数的データ型であることを表わしている．代数的データ型 τ_1，τ_2 が型定義環境 Σ の下での互換であるとは，τ_1 のデータ構築子全てが τ_2 のデータ構築子と互換であることが Σ で宣言されている，もしくは τ_1 と τ_2 がそのような関係の反射的対称的推移的閉包に含まれていることを意味する．

3.3.2 実行時項の型システム

次に実行時項の型システムについて述べる．図 4 は実行時項に対する型付け規則だけ示しており，その他の導出規則はデータ型変換関係をとる以外図 3 と同様である．規則 T_WaitingCheck は待機中検査に対して適用される型付け規則で，検査対象の項 e_2 の型として T を求めることで，e_2 の評価結果が契約 e_1 を満たすかが検査可能であることを表わしている．型付け規則 T_ActiveCheck は活性検査のための規則である．この規則では検査対象の値 v を契約 e_1 に代入すると項 e_2 に評価されることを要求することで，e_2 が検査の中間状態を表わしていることを確認している．型付け規則 T_Forget と T_Exact はそれぞれ型から契約を外す，もしくは型に契約を加えることができることを示している．型に契約を加える際には値 v に対して契約が成り立つことを評価関係を用いて確認している．

最後に，本研究では先行研究 [1] と同様に主部簡約 (subject reduction) を証明するために規則 T_Conv を導入する．この規則の必要性を説明するために関数適用 $e_1\, e_2$ に対する主部簡約について考える．もし e_1 が値であれば，意味論からまず e_2 を評価する．このとき $e_2 \longrightarrow e_2'$ と簡約されたとすると，関数適用全体は $e_1\, e_2 \longrightarrow e_1\, e_2'$ と簡約される．関数適用は T_App によって型付けされるため，e_1 の型を $x{:}T_1 \to T_2$，e_2 の型を T_1 とすると関数適用に付く型は $T_2\{e_2/x\}$ から $T_2\{e_2'/x\}$ に変化する．このとき一般には $T_2\{e_2/x\}$ と $T_2\{e_2'/x\}$ は異なる型であるため，主部簡約が成立するには，このような型を同値なものとして扱う規則が必要となる．型付け規則 T_Conv は型付き項にはそれと同値な型が付くことを許す規則である．型の同値性は次のように定義される．

定義 2 (型同値関係)**.**

- 型定義環境と型上の関係 $\Rrightarrow$ を次のように定義する*:* $\langle \Sigma, \rightarrow \rangle \vdash T_1 \Rrightarrow T_2$ のとき，ある型 T，変数 x，項 e_1 と e_2 について，$T_1 = T\{e_1/x\}$，$T_2 = T\{e_2/x\}$ かつ $\langle \Sigma, \rightarrow \rangle \vdash e_1 \longrightarrow e_2$ が成り立ち，かつ $\Rrightarrow$ はそのような (Σ, T_1, T_2) を全て含む*.*
- 型同値関係 $\equiv$ は関係 $\Rrightarrow$ の対称的推移的閉包である*.*

型健全性を証明するために T_Conv を与えるという点は先行研究と同じであるが，本研究では証明が簡潔になるよう T_Conv で使用している型同値関係の定義を変更している*.*

3.4 型健全性

本研究では λ_μ^H について，弱型健全性 (型付き項は行き詰まり状態にならない) と強型健全性 (型付き項の評価結果は型の意味を保存する) が成り立つことを示す*.* そのためにまず型定義環境の well-formedness を定義する*.*

定義 3.

1. 型定義 $\mathsf{type}\,\tau\,(x{:}T) = \overline{C_i : T_i}^{\,i \in \{1,\ldots,n\}}$ が型定義環境 Σ の下で *well-formed* であるとは，この型定義を ς とすると，次の条件を満たすことをいう*.*
   (a) $\Sigma; \varnothing \vdash T$ が成り立つ*.*
   (b) 任意の $i \in \{1, ..., n\}$ について $\Sigma, \varsigma; x{:}T \vdash T_i$ が成り立つ*.*
2. 型定義 $\mathsf{type}\,\tau\,(x{:}T) = \overline{C_i \parallel D_i : T_i}^{\,i}$ が型定義環境 Σ の下で *well-formed* であるとは，この型定義を ς とすると，次の条件を満たすことをいう*.*
   (a) $\Sigma; \varnothing \vdash T$ が成り立つ*.*
   (b) 任意の $i \in \{1, ..., n\}$ について $\Sigma, \varsigma; x{:}T \vdash T_i$ が成り立つ*.*
   (c) 型定義環境 Σ 中に $\overline{D_i}^{\,i \in \{1,\ldots,n\}}$ 全てが属する代数的データ型が存在する*.*
   (d) 任意の $i \in \{1, ..., n\}$ について，T_i は D_i の引数型と Σ, ς の下で互換である，すなわち $\Sigma, \varsigma \vdash T_i \parallel \mathit{CtrArgOf}_\Sigma(D_i)$ が成り立つ*.*
3. 型定義環境 Σ が *well-formed* であるとは，任意の Σ_1，ς，Σ_2 について，$\Sigma = \Sigma_1, \varsigma, \Sigma_2$ ならば ς は Σ_1 の下で *well-formed* であることをいう*.* このとき $\vdash \Sigma$ と書く*.*

弱型健全性は，先行研究と同様に，主部簡約と進行 (progress) により示すことができる*.*

補題 1 (進行)**.** もし $\langle \Sigma, \rightarrow \rangle; \varnothing \vdash e : T$ ならば次の *(1)–(3)* のいずれかが成り立つ*. (1)* ある項 e' について $\langle \Sigma, \rightarrow \rangle \vdash e \longrightarrow e'$*. (2)* 項 e は値である*. (3)* あるラベル ℓ について $e = \Uparrow\!\ell$*.*

補題 2 (主部簡約)**.** $\vdash \Sigma$ かつ $\langle \Sigma, \rightarrow \rangle; \varnothing \vdash e : T$ かつ $\langle \Sigma, \rightarrow \rangle \vdash e \longrightarrow e'$ ならば $\Sigma; \varnothing \vdash e' : T$*.*

定理 1 (弱型健全性)**.** $\vdash \Sigma$ かつ $\Sigma; \varnothing \vdash e_1 : T$ とする*.* このとき $\langle \Sigma, \rightarrow \rangle \vdash e_1 \longrightarrow^* e_2$ ならば次の *(1)–(3)* のいずれかが成り立つ*. (1)* e_2 は値である*. (2)* あるラベル ℓ について $e_2 = \Uparrow\!\ell$*. (3)* ある項 e_3 が存在し $\langle \Sigma, \rightarrow \rangle \vdash e_2 \longrightarrow e_3$*.*

強型健全性の証明には主部簡約と進行に加え，同値な型は意味論的に等価であることが必要となる*.* そのために，データ型変換関係には同値な項を同じように変換することが求められる*.*

定義 4 (項同値関係)**.**

- 型定義環境と項上の関係 $\Rrightarrow$ を次のように定義する: $\langle\Sigma,\to\rangle \vdash e_1 \Rrightarrow e_2$ のとき，ある型 e，変数 x，項 e_1' と e_2' について，$e_1 = e\{e_1'/x\}$，$e_2 = e\{e_2'/x\}$ かつ $\langle\Sigma,\to\rangle \vdash e_1' \longrightarrow e_2'$ が成り立ち，かつ $\Rrightarrow$ はそのような (Σ, e_1, e_2) を全て含む．
- 項同値関係 $\equiv$ は関係 $\Rrightarrow$ の対称的推移的閉包である．

定義 5. データ型変換関係が型定義環境 Σ の下で同値項を同じように変換するとは，$\langle\Sigma,\to\rangle \vdash e_1 \equiv e_2$ かつ $\Sigma \vdash e_1 \to C\{e_{11}\}e_{12}$ かつ $\Sigma \vdash e_2 \to e_2'$ ならば，ある e_{21} と e_{22} について $e_2' = C\{e_{21}\}e_{22}$ が成り立つことを意味する．またこのようなデータ型変換関係を $\Sigma \vdash_{\equiv} \to$ と書く．

次の補題は同値な型は意味論的に等価であることを表わしている．

補題 3. $\vdash \Sigma$ かつ $\Sigma \vdash_{\equiv} \to$ かつ $\langle\Sigma,\to\rangle \vdash e_1 \Rrightarrow^* e_2$ とすると，$\langle\Sigma,\to\rangle \vdash e_1 \longrightarrow^* \mathsf{true}$ の時かつその時に限り $\langle\Sigma,\to\rangle \vdash e_2 \longrightarrow^* \mathsf{true}$.

定理 2 (強型健全性)**.** $\vdash \Sigma$ かつ $\Sigma \vdash_{\equiv} \to$ とする．このとき $\langle\Sigma,\to\rangle;\varnothing \vdash e_1 : \{x{:}T \mid e_2\}$ かつ $\langle\Sigma,\to\rangle \vdash e_1 \longrightarrow^* v$ ならば $\langle\Sigma,\to\rangle \vdash e_2\{v/x\} \longrightarrow^* \mathsf{true}$.

Proof. 主部簡約と進行により，「$\langle\Sigma,\to\rangle;\varnothing \vdash v : T$ ならば v は T 中の全ての契約を満たす」ことを示せば十分である．これは $\langle\Sigma,\to\rangle;\varnothing \vdash v : T$ の型付け導出木に関する帰納法で証明することができ，特に T_Conv の場合については補題 3 によって示すことができる． □

4 遅延契約検査

第 3 節で与えた λ_μ^H の意味論はキャストを適用するとすぐに実行時検査を行うものであったが，このような実行時検査の正格さは不必要な検査を引き起こすことがあり，特に代数的データ型間のキャストを用いたときにその影響が顕著である．例として第 2 節で与えた代数的データ型 sorted_list′ について考えてみる．代数的データ型 sorted_list′ が付く整数リスト l に対して，リストの先頭要素を取り出す関数 head のような，一般的な型 int list を受け取る関数を適用しようとすると，l に対し sorted_list′ から int list へのキャストを適用することになる．ここで関数 head は受け取った整数リストに対して先頭要素をもつかどうかだけを確認するため，l に head を適用するには l の先頭部分のデータ構築子を Cons に変換すれば十分である．しかし λ_μ^H の意味論では l のデータ構築子を全て Cons または Nil に変換するという不要な処理を行うことになる．

本節では実行時検査においてこのような不要な変換が起こらないように代数的データ型間のキャスト適用を遅延化させるよう λ_μ^H を変更した顕在的契約計算 $\lambda_{\mu_d}^H$ を与える．さらに，λ_μ^H で値になるような項は $\lambda_{\mu_d}^H$ でも値になること，$\lambda_{\mu_d}^H$ で例外を起こすような項は λ_μ^H で停止するならば例外を起こすことを示す．また最後に遅延検査を取り入れた際に予想される問題について考察する．

4.1 計算体系 $\lambda_{\mu_d}^H$ の形式化

計算体系 $\lambda_{\mu_d}^H$ は λ_μ^H の構文，意味論，型システムをそれぞれ変更することで与えられる．以下では λ_μ^H からの変更点についてだけ説明する．

構文　計算体系 $\lambda_{\mu_d}^H$ のプログラム構文は λ_μ^H と同様だが，実行時項として遅延キャスト適用 $\langle\tau_1\{e_1\} \Leftarrow \tau_2\{e_2\}\rangle^\ell @ e$ を加える．これは代数的データ型間のキャスト $\langle\tau_1\{e_1\} \Leftarrow \tau_2\{e_2\}\rangle^\ell$ の項 e への適用を遅延化した状態を表わす．計算体系 $\lambda_{\mu_d}^H$ の構文要素を表わすメタ変数として λ_μ^H のものに添字 d を付けたものを使用すると，具体的な $\lambda_{\mu_d}^H$ の構文は値，項，評価文脈にそれぞれ遅延キャスト適用 $\langle\tau_1\{e_{d1}\} \Leftarrow \tau_2\{e_{d2}\}\rangle^\ell @ v_d$，$\langle\tau_1\{e_{d1}\} \Leftarrow \tau_2\{e_{d2}\}\rangle^\ell @ e_d$，$\langle\tau_1\{e_{d1}\} \Leftarrow \tau_2\{e_{d2}\}\rangle^\ell @ E_d$ を構文要素として加えたものとして与えられる．ただし遅延キャスト適用は実行時に登場する項なので型定義には出現しないものとし，型定義環境のメタ変数は λ_μ^H と同様に Σ を用いる．

意味論　計算体系 $\lambda^H_{\mu_d}$ の評価関係 $\langle \Sigma, \rightarrow_d \rangle \vdash_d e_{d1} \longrightarrow e_{d2}$ を与える．ここで $\rightarrow_d$ は $\lambda^H_{\mu_d}$ におけるデータ型変換関係を表わすメタ変数である．評価関係は代数的データ型間のキャスト適用を遅延させるため，評価関係の導出規則群から E_Eager を除外し，代わりに次に示す評価規則 E_DelayedCast と E_Delayed を導入する．

$$\langle \Sigma, \rightarrow_d \rangle \vdash_d E_d[\langle \tau_1\{e_{d1}\} \Leftarrow \tau_2\{e_{d2}\}\rangle^\ell v_d] \longrightarrow E_d[\langle \tau_1\{e_{d1}\} \Leftarrow \tau_2\{e_{d2}\}\rangle^\ell @ v_d] \quad \text{E\_DelayedCast}$$

$$\frac{\Sigma \vdash \langle \tau_1\{e_{d1}\} \Leftarrow \tau_2\{e_{d2}\}\rangle^\ell C\{e_{d3}\} v_d \rightarrow_d e_d'}{\begin{array}{r}\langle \Sigma, \rightarrow_d \rangle \vdash_d E_d[\mathsf{match}\ \overline{v_d @}\ \langle \tau_1\{e_{d1}\} \Leftarrow \tau_2\{e_{d2}\}\rangle^\ell @ C\{e_{d3}\} v_d\ \mathsf{with}\ \overline{C_i\, x_i \rightarrow e_{di}}^{\,i}] \\ \longrightarrow E_d[\mathsf{match}\ \overline{v_d @}\ e_d'\ \mathsf{with}\ \overline{C_i\, x_i \rightarrow e_d}^{\,i}]\end{array}} \text{E\_Delayed}$$

規則 E_DelayedCast が代数的データ型間キャストの適用を遅延化させ，規則 E_Delayed が遅延キャスト適用が match の対象となったときに実際にそのキャストを適用することを表わしている．規則 E_Delayed における $\overline{v_d @}$ は $\langle \tau'_{11}\{e'_{11}\} \Leftarrow \tau'_{12}\{e'_{12}\}\rangle^{\ell_1} @ ... @ \langle \tau'_{n1}\{e'_{n1}\} \Leftarrow \tau'_{n2}\{e'_{n2}\}\rangle^{\ell_n} @$ を簡略化した表現で，E_Delayed は最初に適用されたキャストから順に実行することを意味する．

型システム　計算体系 $\lambda^H_{\mu_d}$ の実行時項の型付け規則として遅延キャスト適用に対応するものを加える．この型付け規則は，関数適用と似たように，遅延キャスト適用 $\langle \tau_1\{e_1\} \Leftarrow \tau_2\{e_2\}\rangle^\ell @ e$ についてキャスト $\langle \tau_1\{e_1\} \Leftarrow \tau_2\{e_2\}\rangle^\ell$ に関数型が付き，さらにその引数型が e に付くことを求めている．また $\lambda^H_{\mu_d}$ についても，λ^H_μ と同様に，弱型健全性と強型健全性が成り立つ．

4.2　計算体系 λ^H_μ と $\lambda^H_{\mu_d}$ の意味的対応

本節では λ^H_μ と $\lambda^H_{\mu_d}$ におけるプログラムの意味的な関係について述べる．計算体系 λ^H_μ では実行可能な検査はすぐに適用され，$\lambda^H_{\mu_d}$ では必要になった際に適用される．従って λ^H_μ で契約違反が起きなければ $\lambda^H_{\mu_d}$ でも契約違反は起きず，逆に $\lambda^H_{\mu_d}$ で起きた契約違反は λ^H_μ でも起きるはずである．実際に λ^H_μ と $\lambda^H_{\mu_d}$ との間にはこれと似た関係が成り立つが，プログラムの非停止性のために上記のような単純な対応関係とは完全には一致しない．具体的には，λ^H_μ と $\lambda^H_{\mu_d}$ の評価関係について次のような定理が成り立つ．ただし λ^H_μ のプログラムは $\lambda^H_{\mu_d}$ のプログラムでもあること，$\lambda^H_{\mu_d}$ のデータ型変換関係は λ^H_μ のデータ型変換関係でもあることに注意されたい．

定理 3.

1. もし $\langle \Sigma, \rightarrow_d \rangle \vdash e \longrightarrow^* v$ ならば，ある値 v_d が存在し $\langle \Sigma, \rightarrow_d \rangle \vdash_d e \longrightarrow^* v_d$ が成り立つ．さらに $v \in \mathcal{K}_\mathsf{B}$ であれば $v = v_d$ である．
2. $\Sigma; \varnothing \vdash e : T$ が成り立ち e が λ^H_μ の意味論で停止するとする．このとき，$\langle \Sigma, \rightarrow_d \rangle \vdash_d e \longrightarrow^* \Uparrow \ell$ ならばあるラベル ℓ' について $\langle \Sigma, \rightarrow_d \rangle \vdash e \longrightarrow^* \Uparrow \ell'$ が成り立つ．

4.3　プログラミング言語への遅延契約検査導入に向けて

遅延契約検査には，実際には契約に違反するようなデータ構造であったとしても契約違反が発見されないという問題がある．例えば整数リストに sorted_list′ への遅延キャストを適用し，その先頭要素を取り出すとする．このとき先頭要素が二番目の要素より小さいことはすぐさま検査されるが，残りの部分リストが昇順整列リストであるかの検査は遅延される．直観的には sorted_list′ の先頭要素はリストの最小要素であるが，それは部分リストに関する検査が成功して初めて確定する．そのため，残りの部分リストにアクセスすることなく，リストの先頭要素を最小要素とみなすことはバグを引き起こす可能性がある．

従って遅延契約検査をプログラミング言語に取り入れる際には，何らかの制限を課すことが必要だと考えられる．以下では具体的な導入案をいくつか述べる．

- Racket 言語などと同様[3]に，契約検査を正格に行うか遅延させて行うかを選択可能にする．
- システムが許可した契約検査だけ遅延させることを許す．例えば sorted_list′ のデータ構築子 SCons は関数 `nil`，`head` を参照していたが，これらはリストの関数であるため，適用するためには sorted_list′ からリストへ再変換する必要がある．このような場合，sorted_list′ からリストへの変換の遅延化を許すことで不必要な変換を避けることができる．これは特に greater_list などの int list 以外のリスト構造から sorted_list′ へ変換する際に有効だと考えられる．
- Chitil [3] のように，構成要素間の関係を表現した代数的データ型を許可しない．

5 実装

今回はキャストが追加された拡張 OCaml コードを処理できるプリプロセッサを実装した．キャストには型名 τ，篩型 $\{x{:}T \mid e\}$，依存関数型 $x{:}T_1 \to T_2$，n 項に一般化した依存積型 $x_1{:}T_1 \times \ldots x_{n-1}{:}T_{n-1} \times T_n$ が使用できる．型定義には新しい型定義構文を追加し，キャストに使用できるユーザー定義型は新しい構文を使って定義されたもののみとする．

実装には Camlp4 を用いた．Camlp4 は OCaml 向けのプリプロセッサである．また，Camlp4 に備わっている機能を用いることで，Camlp4 内のパーサの動作を変更するプログラムを書くことができる．これを用いてキャスト，契約を含んだ型，新しい型定義をパースするエントリを追加し，拡張 OCaml コード中のキャストを，キャストの動作をシミュレートする OCaml コードへ変換する．

5.1 変換規則の概略

キャストは契約の検査を行う関数へ変換される．例えば $\langle \{x{:}\mathsf{int} \mid e\} \Leftarrow \mathsf{int} \rangle^{\ell}$ は以下のように変換される．$\Uparrow\ell$ は ℓ を文字列として例外 `Blame` ℓを起こすことでシミュレートされる．

```
fun x -> if e then x else raise (Blame ℓ)
```

依存関数型間のキャスト $\langle x_1{:}T_{11} \to T_{12} \Leftarrow x_2{:}T_{21} \to T_{22} \rangle^{\ell}$ は以下のように変換される．

```
fun _f _x -> let x1 = _x and x2 = ⟨T21 ⇐ T11⟩^ℓ _x in ⟨T12 ⇐ T22⟩^ℓ (_f x2)
```

変換後のコード中に現れるキャストは OCaml コードになるまで再帰的に変換される．変数 x_1, x_2 のスコープに現れる型が依存関数型の意図通り，T_{12} と T_{22} だけであることに注意する．アンダースコアで始まる名前を持つ変数は新たに導入される変数である．ソースコード中に現れる変数名との衝突を防ぐためにアンダースコアで始まる変数名は予約語とした．依存積型のキャストについても変数の束縛関係に注意しながら各要素をキャストすることでほぼ同様に実現される．

代数的データ型間のキャスト $\langle \tau_2 \Leftarrow \tau_1 \rangle^{\ell}$ では，まず τ_1 の各データ構築子 C_i と互換関係にある τ_2 のデータ構築子 $D_{i\,1}, \ldots, D_{im}$ を型情報から得る．得られた情報を元にしてキャストは以下のように変換される．

```
fun _v -> let rec _τ1_τ2 = function
            | C1 _v -> try D11 (⟨T11 ⇐ T211⟩^ℓ _v) with Blame _ ->
                       try D12 (⟨T11 ⇐ T212⟩^ℓ _v) with Blame _ ->
                       ...
            | ...
          in _τ1_τ2 _v
```

引数型間のキャスト (例えば $\langle T_{11} \Leftarrow T_{211} \rangle^{\ell}$) が成功するようなデータ構築子が見つかるまで順に試していく．一旦再帰関数へ変換してから呼び直しているのは，再帰的に定義された型のキャストに対応するためである．キャストを再帰的に変換していく中で，再帰関数へと変換されたキャストが

[3] http://docs.racket-lang.org/reference/data-structure-contracts.html

再び現れた場合は変換を行わず，単に再帰関数名で置き換えることによって無限に変換が繰り返されることを防ぐ．代数的データ型が type $\tau(x)$ のように抽象化されて定義されている場合は，let 式で x を適用する項の値に束縛してから let 式の本体にキャストの変換結果を挿入する．

新しく追加した構文での型定義は契約に関する部分の表記を落として，OCaml に元々ある構文を使った型定義へ変換される．例えば sorted_list′ の型定義は以下のように変換される．

```
type sorted_list' =
  SNil  of unit
| SCons of int * sorted_list'
```

変換と同時に変換前のソースコードに書いてあった契約についての情報は型情報として記録し，定義した代数的データ型を含んだキャストを変換する際に利用する．

5.2 実装時の問題とその解決

実装では大きな問題が二つ問題が生じた．一つ目は変数の束縛回避の問題である．例えば $\langle x_1{:}\mathsf{int} \to \{y{:}\mathsf{int} \mid e_1\} \Leftarrow x_2{:}\mathsf{int} \to \{y{:}\mathsf{int} \mid e_2\}\rangle^{\ell}$ というキャストを変換すると x_1 のスコープに $\{y{:}\mathsf{int} \mid e_2\}$ が入る．もし x_1 が e_2 に自由に現れていると e_2 中の x_1 が本来意図した値とは別の値に束縛されてしまう．実装では束縛回避が必要となる条件を洗い出し，変数名の置き換えを行うことで解決した．

二つ目は定義しようとしている型を含むキャストが定義中に現れるような型定義が直接変換できない問題である．これは実装に Camlp4 を使用していることに起因するもので，契約を表わす述語を OCaml の式として型定義情報に記録することが原因である．述語中に現れるキャストは変換してから記録しなければならないが，変換するためには現在定義中の型情報が必要であり変換が行えない．この問題は述語部分に現れるキャストを文字列へ変換することで解決した．一時的に文字列を値に適用するようなコードへ変換されるが，Camlp4 は型検査を行わないのでプリプロセス中は正しい式として扱える．型情報にはキャストを文字列化した述語を記録しておき，定義型を含むキャストを変換する時点で述語中の文字列を再度変換パーサに通して最終的な OCaml コードへ変換する．

5.3 実行時契約検査の最適化に向けて

データ構造の実行時検査にかかる時間計算量は，構成要素をどのような順番で検査するかによって大きく異なることがある．例として，要素がある整数 n から 0 まで順番に現われ，0 の次にはまた n が現われるような整数リストを考えてみよう．これは代数的データ型では次のように定義できる．

```
type circular_list (m:int) (n:int) =
  CNil  of unit
| ZCons of {x:int|m = x && m <= 0} * circular_list n n
| CCons of {x:int|m = x && m > 0}  * circular_list (m-1) n
```

整数 m はリストの先頭要素と等しい整数を表わしている．このとき構成要素の検査を再帰的データ構造から行うと，整数リストから `circular_list` へのキャストにかかる最悪時間計算量のオーダーは，リストの長さを l とすると，$O(2^l)$ となる．一方先頭要素に関する検査から行うと，最悪時間計算量オーダーは $O(l)$ となる．

構成要素の実行時検査に関する計算量の増大を抑えるための最適化として，型の意味が他のデータ構築子と最も異なる部分から検査したり，再帰的データ構造の検査を後回しにすることなどが考えられる．また，篩型中で用いるデータ構造には遅延検査を行うのも有用であるかもしれない．

6 関連研究

一般に篩型は，整数型などの単純な型に細かな条件を課すことで，0 より大きな整数といった，より正確な意味をもつ型を記述するための機構である．型を項の集合と見たとき，篩型は条件を課す

前の型の部分集合と解釈できることから部分集合型 (subset type) とも呼ばれる [17, 2]．篩型はデータ構造に関する仕様を型システムで表現するために Freeman ら [10] によって提案された．Freeman らの研究では，篩型は代数的データ型の部分型として与えられ，適用できるデータ構築子を指定することで構築可能なデータ構造の形を制限する．しかし，型検査や型推論アルゴリズムを決定可能にするため，データ構造の仕様を述語 (契約) として与えることはできなかった．

Flanagan ら [9, 13] は篩型を用いて表わした仕様 (契約) を静的検査と実行時検査を組み合わせることで検証する機構を備えた計算体系として，顕在的契約計算体系を与えた．Belo ら [1] は顕在的契約計算体系の型システムを構文的操作だけで与える方法を考案し，実際にそれを用いて全称型で拡張した計算体系を与えた．また Knowles らは Flanagan らの顕在的契約計算を動的型や第一級型で拡張したプログラミング言語 Sage [14] を開発した．Sage ではプログラマが代数的データ型を定義することができ，本研究と同様に，データ構築子に契約を付与したり項変数の抽象化が可能である．ただし，Sage の実装では代数的データ型はチャーチエンコーディングにより実現されており，代数的データ型を含んだ計算体系はエンコーディング方法も含めて形式化されていない．また代数的データ型が表現する契約の検査を，特にデータ構築子の数が異なる場合に，どのように行うかも明らかにしていない．

篩型が表わす仕様をプログラムが満たすかを全て静的に検査する研究としては DML [18] や liquid type [12, 16] などの静的解析技法や Coq などの定理証明支援系が挙げられる．DML は仕様記述言語を制限することで依存型を備えた実践的なプログラミング言語として，Xi によって提案された．Jhala らは liquid type と呼ばれる機構を用いて，本研究で扱ったような契約情報を取り入れた代数的データ型を含む計算体系のための型推論アルゴリズムを提案した．Coq などのいくつかの定理証明支援系では代数的データ型に命題という形で仕様を与えることができるが，プログラム片がその仕様を満たすことを示すにはプログラマが証明を与える必要がある．

またデータ構造の契約を実行時検査するための研究もいくつか為されている [11, 3, 4]．Hinze ら [11] は静的型付き高階関数型プログラミング言語に契約機構を与える方法について論じた．Hinze らはデータ構造に対する契約コンビネータも与えたが，このコンビネータはデータ構造の構成要素間の契約は記述できなかった．Chitil [4] は Haskell のように必要呼び意味論をもつ関数型プログラミング言語での契約機構について考察し，ライブラリとして実装した．また Chitil は必要呼び意味論における契約の表示的意味論を与え，データ構造の要素間の関係に言及する契約は記述できないことを示した [3]．

Findler ら [8] は意味論が正格なプログラミング言語で実行時契約検査を遅延させる契約機構の実装技法について論じ，さらに契約の正格検査は，遅延検査に比べ，重大な計算コストを要する場合があることを示した．また Degen ら [5] は名前呼び (または値呼び) 意味論をもつ計算体系に契約の正格，遅延検査 (または正格検査) の仕組みを与え，その性質について論じた．しかし値呼び意味論をもち，契約を遅延検査する計算体系は考慮しなかった．

7　おわりに

顕在的契約計算はソフトウェア契約をできるだけ静的に検査し，静的検査が難しいものについて実行時に検査する．この計算体系では契約は篩型として型システムに取り込まれているが，データ構造上の契約を表現するには篩型では不十分であった．本論文ではそのような契約を効果的に表現できる計算体系として，代数的データ型を加えた顕在的契約計算 λ_μ^H を与えた．本研究で扱った代数的データ型は，データ構築子への契約の付与や項変数の抽象化により様々な契約情報を効果的に表わすことができるだけでなく，データ構造の構成要素間の不変条件を実行時に検査するための機構も備えている．さらに本研究では，λ_μ^H の性質について論じただけでなく，λ_μ^H のプログラムを既存のプログラミング言語のものへ変換することで λ_μ^H に基づくプログラミング言語の実装を試みた．

本研究の将来的な課題はいくつか考えられる．まず初めに，λ_μ^H で契約の静的検査を行うための機構を与えるという課題がある．本論文では代数的データ型を加えた顕在的契約計算における契約の実行時検査機構は与えたが，静的検査のための仕組みは与えていない．契約の静的検査を実現するためには，Belo ら [1] と同様に，部分型関係を定義し，その部分型関係が「妥当」であることを示すことになる．ただし λ_μ^H では代数的データ型に対する部分型関係を与える必要があるため，先行研究の方法をそのまま λ_μ^H に適用することはできないだろうと考えられる．次に，本研究では計算体系の意味論をデータ型変換関係を一つ定めることで与えたが，λ_μ^H や $\lambda_{\mu_d}^H$ をプログラミング言語として実装する際にどのようなデータ型変換関係を用いるかということも将来の課題である．この問題は根本的には解決できず，いくつかのトレードオフのある解法から目的に合うものを選択することになるのかもしれない．最後に，データ構造に関する篩型と代数的データ型はどちらもデータ構造の仕様を表わすが，これらの間にどのような関係が成り立つかを調査することも課題として残されている．具体的には，契約情報が何らかの形で制限されていれば，データ構造に関する篩型から同じ意味の代数的データ型を機械的に導くことが可能だと予想しており，この結果を用いると契約付きの代数的データ型の記述がより容易になると考えられる．

謝辞　有益なコメントをして下さった PPL の査読者に感謝する．

参考文献

[1] J. F. Belo, M. Greenberg, A. Igarashi, and B. C. Pierce. Polymorphic contracts. In *Proc. of ESOP*, volume 6602 of *LNCS*, pages 18–37, 2011. Springer-Verlag.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

[3] O. Chitil. A semantics for lazy assertions. In *Proc. of ACM PEPM*, pages 141–150, 2011. ACM.

[4] O. Chitil. Practical typed lazy contracts. In *Proc. of ACM ICFP*, pages 67–76, 2012. ACM.

[5] M. Degen, P. Thiemann, and S. Wehr. Eager and delayed contract monitoring for call-by-value and call-by-name evaluation. *J. Log. Algebr. Program.*, 79(7):515–549, 2010.

[6] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

[7] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proc. of ACM ICFP*, volume 37, pages 48–59, 2002. ACM.

[8] R. B. Findler, S. Guo, and A. Rogers. Lazy contract checking for immutable data structures. In *Proc. of IFL*, volume 5083 of *LNCS*, pages 111–128, 2008. Springer-Verlag.

[9] C. Flanagan. Hybrid type checking. In *Proc. of ACM POPL*, pages 245–256, 2006. ACM.

[10] T. Freeman and F. Pfenning. Refinement types for ML. In *Proc. of ACM PLDI*, pages 268–277, 1991. ACM.

[11] R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In *Proc. of FLOPS*, volume 3945 of *LNCS*, pages 208–225, 2006. Springer-Verlag.

[12] M. Kawaguchi, P. M. Rondon, and R. Jhala. Type-based data structure verification. In *Proc. of ACM PLDI*, pages 304–315, 2009. ACM.

[13] K. Knowles and C. Flanagan. Hybrid type checking. *ACM TOPLAS*, 32(2:6):1–34, 2010.

[14] K. Knowles, A. Tomb, J. Gronski, S. N. Freund, and C. Flanagan. Sage: Unified hybrid checking for first-class types, general refinement types, and dynamic (extended report). Technical report, 2007. `http://sage.soe.ucsc.edu/sage-tr.pdf`.

[15] B. Meyer. *Object-Oriented Software Construction, 1st Editon*. Prentice-Hall, 1988.

[16] N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *Proc. of ESOP*, volume 7792 of *LNCS*, pages 209–228, 2013. Springer-Verlag.

[17] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proc. of ESOP*, volume 5502 of *LNCS*, pages 1–16, 2009. Springer-Verlag.

[18] H. Xi. Dependent ML: An approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, 2007.

[19] D. N. Xu. Hybrid contract checking via symbolic simplification. In *Proc. of ACM PEPM*, pages 107–116, 2012. ACM.

[20] D. N. Xu, S. L. Peyton Jones, and K. Claessen. Static contract checking for Haskell. In *Proc. of ACM POPL*, pages 41–52, 2009. ACM.