高階契約を持つプログラミング言語に対するトレース意味論

村井 涼　　五十嵐 淳　　中澤 巧爾

京都大学大学院情報学研究科

{ rmurai@fos.kuis, igarashi@kuis, knak@kuis } .kyoto-u.ac.jp

概 要　本論文は，Racket 流の高階契約を持つプログラミング言語に対するトレース意味論を提示する．一般に，トレース意味論は，プログラムの意味をプログラム自身とそのプログラムが使われる文脈との間の可能な相互作用の列の集合（「トレース集合」）として与える．われわれのトレース意味論は，それに加えて関数のサーバ側モジュールとクライアント側モジュールの境界で行われる実行時契約検査を明示的に表現しており，これまで提案されてきたものより契約検査の本質が見て取りやすい意味論になっている．

また，われわれはこのトレース意味論が一般的な操作的意味論と整合的であることを保証する二つの定理を得ている．われわれのトレース意味論においては，(1) 異なるモジュール群間の相補的な二つのトレースが操作的意味論における簡約列と対応する．さらにこのことを用いて，(2) トレース意味論が文脈等価性に対して健全である，つまりトレース集合の等しさが文脈等価性を含意することも示した．また，われわれは (2) の逆も成立すること，つまりトレース意味論が完全抽象であることを予想している．

1　はじめに

Eiffel や Racket といったプログラミング言語では，プログラマはモジュール内で実装した関数の仕様として「ソフトウェア契約」（以下単に「契約」と呼ぶ）をインタフェースに書くことができる．契約は事前条件と事後条件の二つからなる．前者はその関数を呼び出すモジュールがみたすべき，実引数に関する条件であり，後者はその関数を呼び出されたモジュールがみたすべき，返り値に関する条件である．これらの条件がみたされるかは実行時に検査される．検査により，条件をみたせなかった，つまり契約を破ったモジュールが特定され，ブレーム (blame) と呼ばれる例外としてプログラマに知らされる．これによってバグの特定が容易になる．

契約の概念は，Findler ら [6] によって，高階関数に対する契約—高階契約— として拡張され，現在に至るまで活発に研究がなされている．その中のトピックの一つとして「ブレーム割り当ての正しさ」という問題がある．この問題を扱った [3] では契約検査によるブレーム割り当てが正しく責任のあるモジュールを特定しているかを議論する形式的な枠組みを提供している．しかし，この枠組みでは，契約検査による値の監視を,「一般的な式に契約とその関係者を付した式」の簡約によって表現しており,「モジュールの境界を超える値のやりとりの監視」という契約検査の本質を反映していないという意味で複雑なものになっている．

本論文はこの問題を「トレース意味論」と呼ばれる手法によって，解決しようとする．一般に，トレース意味論は，プログラムの意味をプログラム自身とそのプログラムが使われる文脈との間の可能な相互作用を表すイベント列の集合（「トレース集合」）として与える．われわれのトレース意味論は，それに加えてサーバモジュール群とクライアントモジュール群の境界で行われる実行時契約検査を明示的に表現しており，契約検査の本質が見て取りやすい意味論になっている．

また，われわれのトレース意味論は従来与えられてきた操作的意味論に照らして妥当なものである．われわれのトレース意味論においては，(1) 二つのモジュール群を異なる計算主体とみなしたときの両者間の値の通信を表現している二つの双対なトレースと，その二つのモジュール群の結合にもとづく操作的意味論における簡約列とが対応する（補題 6.1）．さらにこのことを用いて，(2)

トレース意味論が文脈等価性に対して健全である，つまりトレース集合の等しさが文脈等価性を含意すること（定理 6.1）も示した．これらの事実はわれわれの提示したトレース意味論が [6] で定義されたような従来の操作的意味論と整合することを保証している．また，われわれは (2) の逆も成立すること，つまりトレース意味論が文脈等価性に対して完全抽象であることを予想しており，重要な補題となる定義可能性 (definibaility) を仮定した上での証明を提示した．

次節以降の構成を簡単に説明する．第 2 節では，対象とする言語の構文と操作的意味論について概説する．第 3 節では，型システムを導入し，保存と進行の定理を示す．第 4 節では，ライブラリのように部品として使えるモジュール群を「コンポーネント」として定義し，コンポーネントを単位として文脈等価性を定義する．第 5 節では，コンポーネントとそれが使われる文脈との間で行われる可能な値のやりとりを表現するラベル付き遷移系 (LTS) を用いてトレース意味論を定義し，その利点を述べる．第 6 節では，第 4 節と第 5 節の定義に従って双対トレースと簡約列の対応を表す性質や健全性を証明する．後半では定義可能性を予想として立て，これを用いて完全抽象性が示せることを確認する．第 7 節では，本研究の関連研究について述べ，第 8 節で結論を述べる．

2　構文と操作的意味論

まず，われわれが研究対象とするプログラミング言語の構文と操作的意味論について概説する．定義を図 1 と図 2 に示すが，規則を説明する前に，まずこの言語の特徴を簡単に述べる．

1. 関数はすべてどこかのモジュールで契約を付されて定義され，$m.f$ のかたちで参照される．
2. 契約は基底型（本論文では nat 型のみ）に対する契約（「基底契約」と呼ぶ．通常，これはプログラミング言語で記述されるが単純化のため値の部分集合を指定する）と関数契約コンストラクタ $\mapsto$ から構成する．$\mapsto$ の左側を「引数契約」，右側を「返り値契約」と呼ぶ．$\mapsto$ を入れ子にすることで，高階関数に対する契約が記述できる．契約検査は基底型の値が，(1) 関数に渡されるとき，(2) 関数呼び出しの結果として呼び出し元に返されるときに行われる [1]．契約違反発生の事実とその違反者 m を $\Uparrow_m$ で表現し，これを「m へのブレーム」と呼ぶ．
3. プログラムを実行するために用意された特別なモジュール $main$ が存在する．計算は $main$ ではじまり，他のモジュールの関数を呼び出しながら進行し，(契約違反による実行の中断がなければ）$main$ で計算結果の値が得られる．
4. 高階関数に関数値を引数として渡したり，関数呼び出しの返り値として関数値を返したりする際は，その関数に新たに関数名をつけてその名前をやりとりする．高階関数に与えられた (高階) 契約の引数/返り値部をその名前の契約として引き継ぐことにより，Findler ら [6] の高階契約検査の機構と同等な契約検査を行っている．名前と関数値の対応関係はモジュール群 M を拡張していくことによって管理される．
5. 各モジュールにはそのモジュールのみが利用できる，書き換え可能な変数 a が用意されている．$!a$ が変数からの読み出し，$a := e_1; e_2$ が a に e_1 の値を書き込んだ後 e_2 を実行するための構文である．これらの変数の内容はストア μ で管理されている．変数に割り当てられるのは単純化のため基底型の値（すなわちここでは自然数）のみとする．
6. 式 e が評価されるモジュールと評価された結果得られる値がみたすべき契約 c を明示するための表現 $[e]^c_m$ が存在する．これらの情報は契約検査を行う際に参照される．

特徴 4 についてこのような構成にした意図を簡単に説明する．モジュール間の関数値のやりとりを名前を付けて行うことは，実行上なんら本質的な役割を果たさないので一見冗長に見えるが，これは完全性をみたすトレース意味論を構成するために必要と思われる措置である．後の章で与えら

[1] (1) で関数の事前条件が，(2) で関数の事後条件がチェックされる．

$$
\begin{array}{rcll}
m, m', \ldots & \in & \mathit{MName} = \{\mathit{main}, \ldots\} & \text{(モジュール名)} \\
f, g, \ldots & \in & \mathit{FName} \quad \text{(関数名)} \qquad a \in \mathit{AName} & \text{(書き換え可能変数)} \\
c & \in & \mathit{Contracts} ::= P\ (\subseteq \mathbb{N}) \mid c \mapsto c & \text{(関数に付される契約)} \\
op & & ::= + \mid * & \text{(算術演算)} \\
e & \in & \mathit{Exp} ::= x\ (\in \mathit{Var}) \mid \underline{n}\ (n \in \mathbb{N}) \mid e\ op\ e \mid ee & \text{(式)} \\
 & & \quad \mid (\mathsf{case}\ e\ \mathsf{of}\ 0 \mapsto e \mid S(x) \mapsto e) \mid m.f \mid [e]^c_m \mid (a := e; e) \mid\ !a & \\
v & \in & \mathit{Val} ::= \underline{n} \mid m.f & \text{(値)} \\
\mathit{impl} & \in & \mathit{Impl} ::= m.f \mid \lambda \vec{x}.e & \text{(関数の実装)} \\
M & \in & \mathit{MName} \xrightarrow{fin} \mathit{FName} \xrightarrow{fin} \mathit{Contracts} \times \mathit{Impl} & \text{(モジュール群)}
\end{array}
$$

図 1. 言語の構文

れる概念を先取りして以下でもう少し説明する．第 5 節で定義されるトレースに契約の情報が現れると，文脈等価性を反映する適切な抽象度のトレース集合を定義するのが難しくなるという問題がある．しかし，以下で見るように，モジュールをまたぐ関数値に関する契約は常にモジュール内部で保持しその関数値に付けられた名前だけをやりとりするようにすればトレースには契約の情報が現れず，完全性をみたすに十分な抽象度のトレース集合が得られると考えられる．

ここで，表記上の注意をいくつか与える．M は図 1 に示したように定義域が有限の部分関数として定義されている．すなわち，定義域 $dom(M)$ は M が含むモジュールの名前の集合を表す．本論文では，この関数をそのグラフ集合と同一視し，この集合の要素を $m.f/c = impl$ という形式で記述する．また，$M(m.f)$ で $M(m)(f)$ を表すとする．またこうした関数のアップデートを記号+によって表す．例えば，$M + (m.f/c = impl)$ は，M に $m.f/c = impl$ を追加したモジュール群である．さらに，モジュール群の実装を隠す操作を $\mathit{Intf}(M) = \{m.f/c \mid (m.f/c = impl) \in M\}$ と定義する．

次に，上で挙げた特徴を中心に操作的意味論（図 2）について説明する．操作的意味論は，式 e，モジュール群 M，ストア μ の組に対する，簡約が起きるモジュールの名前を明示した簡約関係によって定義される．ここで，ストア μ はモジュール名と書き換え可能変数に対する定義域有限の部分関数であり，記法はモジュール群に関するものと同様の記法を用いる．契約検査の結果次第で右辺は左辺と同じ形式の三つ組となったり，契約違反を起こしたモジュール m へのブレーム $\Uparrow_m$ となったりする．ブレームは一種の例外であり，またこれをキャッチして何らかの処理を行うような機構はわれわれの言語には存在しないから，上記特徴 3 の括弧内で示唆したとおり，ブレームの発生は実行の中断を意味する．このことは合同規則 (2-A)，(2-B) で表現されている．ここで，規則を A と B に分けているのは，A では規則の結論部と前提部で実行されるモジュールが異なる一方，B では同一であるからである．これは (1-A)，(1-B) においても同様である．

関数適用の規則は引数の種類によって大きく二つに分かれ（(AppN)，(AppF)），適用される関数の実装の種類によってさらにそれぞれ二つに分かれる（-1，-2）．(AppN) では特徴 2 の (1) の場合が実現されている．契約検査を通過する場合，(AppN1) では $m'.f$ の実装 $\lambda\vec{x}.e$ における $\vec{x}$ の長さに応じてさらに場合分けがされる．長さが 2 以上の場合，適用の結果は依然として関数であるが，特徴 1，4 で述べたとおり，この部分適用の結果の関数につけられた名前 $m'.f'$ が簡約の結果となり，その名前と指示対象が M に記憶される．$\vec{x}$ の長さが 1 の場合，特徴 6 で述べた形式の $[e[n/x]]^c_{m'}$ が簡約結果となる．これは $m'.f$ の本体 (body) に実引数を代入した式 $e[n/x]$ の簡約はモジュール m' において行われること，その評価結果が c をみたすことに m' が責任を負うことを示している．前者は合同規則 (1-A)，(2-A) で，後者は規則 (ResN)，(ResF) で表現されている．(AppN2) は実装が関数名となっているだけで，本質的には (AppN1) と同じである．

規則 (AppF) も場合分けの仕方は基本的に (AppN) と変わらないが，契約検査が行われない点，引数に名前が付けられて名前が代入される点が (AppN) と異なっている．(AppF1) の一番目のケー

$$
\begin{array}{rcll}
\mu & \in & \mathit{MName} \xrightarrow{fin} \mathit{AName} \xrightarrow{fin} \{\underline{n} \mid n \in \mathbb{N}\} & (\text{ストア}) \\
E & ::= & [] \mid E\ op\ e \mid \underline{n}\ op\ E \mid (\mathsf{case}\ E\ \mathsf{of}\ 0 \mapsto e \mid S(x) \mapsto e) & (\text{評価文脈}) \\
 & & \mid Ee \mid (m.f)E \mid [E]^c_m \mid a_i := E; e &
\end{array}
$$

$\boxed{e; M; \mu \to_m e'; M'; \mu'}$ $\boxed{e; M; \mu \to_m \Uparrow_{m'}}$

$$\frac{(n_1\ op\ n_2 = n_3)}{\underline{n_1}\ op\ \underline{n_2}; M; \mu \to_m \underline{n_3}; M; \mu}\ (\textsc{Arith}) \qquad \frac{}{\mathsf{case}\ \underline{0}\ \mathsf{of}\ 0 \mapsto e \mid S(x) \mapsto e'; M; \mu \to_m e; M; \mu}\ (\textsc{Case-1})$$

$$\frac{(n > 0)}{\mathsf{case}\ \underline{n}\ \mathsf{of}\ 0 \mapsto e \mid S(x) \mapsto e'; M; \mu \to_m e'[\underline{n-1}/x]; M; \mu}\ (\textsc{Case-2})$$

$$\frac{(m'.f/(P \mapsto c) = \lambda\vec{x}.e) \in M}{(m'.f)\underline{n}; M; \mu \to_m \begin{cases} m'.f'; M + (m'.f'/c = \lambda x_2 ... x_i.e[\underline{n}/x_1]); \mu & \\ \qquad (n \in P,\ \vec{x} = x_1 ... x_i\ (i > 1),\ f' \text{ is fresh}) & \\ [e[\underline{n}/x]]^c_{m'}; M; \mu & (n \in P,\ \vec{x} = x) \\ \Uparrow_m & (n \notin P) \end{cases}}\ (\textsc{AppN-1})$$

$$\frac{(m'.f/(P \mapsto c) = m''.g) \in M}{(m'.f)\underline{n}; M; \mu \to_m \begin{cases} [(m''.g)\underline{n}]^c_{m'}; M; \mu & (n \in P) \\ \Uparrow_m & (n \notin P) \end{cases}}\ (\textsc{AppN-2})$$

$$\frac{(m'.f/(c_1 \mapsto c_2) \mapsto c_3 = \lambda\vec{x}.e) \in M}{(m'.f)(m''.g); M; \mu \to_m \begin{cases} m'.f'; M + (m'.f'/c_3 = \lambda x_2 ... x_i.e[m.g'/x_1]) + (m.g'/c_1 \mapsto c_2 = m''.g); \mu & \\ \qquad (\vec{x} = x_1 ... x_i\ (i > 1),\ f' \text{ and } g' \text{ are fresh}) & \\ [e[m.g'/x]]^{c_3}_{m'}; M + (m.g'/c_1 \mapsto c_2 = m''.g); \mu & (\vec{x} = x,\ g' \text{ is fresh}) \end{cases}}\ (\textsc{AppF-1})$$

$$\frac{(m'.f/(c_1 \mapsto c_2) \mapsto c_3 = m'''.h) \in M \qquad (g' \text{ is fresh})}{(m'.f)(m''.g); M; \mu \to_m [(m'''.h)(m.g')]^{c_3}_{m'}; M + (m.g'/c_1 \mapsto c_2 = m''.g); \mu}\ (\textsc{AppF-2})$$

$$\frac{}{[\underline{n}]^P_{m'}; M; \mu \to_m \begin{cases} \underline{n}; M; \mu & (n \in P) \\ \Uparrow_{m'} & (n \notin P) \end{cases}}\ (\textsc{ResN}) \qquad \frac{(f' \text{ is fresh})}{\begin{array}{l}[m''.f]^c_{m'}; M; \mu \\ \to_m m'.f'; M + (m'.f'/c = m''.f); \mu\end{array}}\ (\textsc{ResF})$$

$$\frac{}{(a := \underline{n}; e); M; \mu \to_m e; M; \mu + (m.a = \underline{n})}\ (\textsc{Assg}) \qquad \frac{}{!a; M; \mu \to_m \mu(m.a); M; \mu}\ (\textsc{Deref})$$

$$\frac{e; M; \mu \to_{m'} e'; M'; \mu'}{[e]^c_{m'}; M; \mu \to_m [e']^c_{m'}; M'; \mu'}\ (1\text{-A}) \qquad \frac{\begin{array}{c}e; M; \mu \to_m e'; M'; \mu' \\ E = E'[[E'']^c_m] \text{ なる } E',\ E'' \text{ は存在しない}\end{array}}{E[e]; M; \mu \to_m E[e']; M'; \mu'}\ (1\text{-B})$$

$$\frac{e; M; \mu \to_{m'} \Uparrow_{m''}}{[e]^c_{m'}; M; \mu \to_m \Uparrow_{m''}}\ (2\text{-A}) \qquad \frac{\begin{array}{c}e; M; \mu \to_m \Uparrow_{m'} \\ E = E'[[E'']^c_m] \text{ なる } E',\ E'' \text{ は存在しない}\end{array}}{E[e]; M; \mu \to_m \Uparrow_{m'}}\ (2\text{-B})$$

図 2. 操作的意味論

スを例にこの点について意図を説明する．部分適用の結果に $m'.f'$ という名前がつけられそれが簡約結果となっているのは，(AppN1) と変わらない．しかし $m'.f'$ の実装を見ると e に代入されているのは $m''.g$ ではなく新たに生成された名前 $m.g'$ である．これには新たに契約 $c_1 \mapsto c_2$ が課せられているが，その実体は $m''.g$ である．ここで m'' の関数をわざわざ m のものとして名付け直したのは $m'.f$ の引数がみたすべき契約 $c_1 \mapsto c_2$ を関数適用を行ったモジュール m に課したいからである．ただし実際には $m.g'$ が $c_1 \mapsto c_2$ をみたすかどうかが直接検査されるわけではない．c_1 は $m.g'$ が使われたとき，使う側であるモジュール m' に課され， c_2 は $m.g'$ が使われたときの計算結果の値に課される．当然 c_1 や c_2 も関数契約のときは (AppF) や (ResF) にもとづいて契約が実引数や返り値に引き継がれるので，結局引数契約や返り値契約が関数契約の場合，実際の契約検査はこうした過程を経て基底契約に行きつくまで遅延される．このようなしくみは本質的に [6] で導入された高階契約計算とまったく同等である．

特徴 5 を実現するのが (Assg) と (Deref) である．これらの規則は通常と同じようにストアのアップデートとストアからの値の取り出しを用いて定義される．ML でいう ref のような構文はないので，ストアの定義域は不変であることに注意されたい．とくに，ストアのアップデート $\mu + (m.a = \underline{n})$ は，$\mu(m.a)$ が既に定義されているような μ に対してのみ定義されている操作であるとする．

(Arith) は算術演算を行う規則であり，(Case) は case 式を使って与えられた自然数が 0 であるか否かで条件分岐させるための規則である．

3 型システム

本節ではわれわれの言語に対する型システムについて説明する．図 3 に型システムの定義を示す．

われわれの型システムは，基本的に [12] にあるような参照つき単純型付きラムダ計算に対するそれと変わらない．また契約に対する型付けも [6] におけるそれと同様，ごく自然なかたちで与えてある． この型システムは，直観的には，プログラマが実装するモジュール群 M のほかに，外部から提供されるモジュール群の型情報 Σ をもとに，式の型整合性を判定するものである．すなわち，M と Σ の関数名と契約の情報，自由変数の型を指定する型環境 Γ，そしてストア μ 内の値の型を指定する S をもとに式 e に型がつくかを判定するものである．ここでは Σ をシグネチャ群と呼び，モジュール名と関数名に対してその関数の契約を返す，定義域有限の部分関数として定義する．型付けのしかたを見ると，われわれの言語では型は契約から一意に定まることがわかる．したがって $dom(M)$ と $dom(\Sigma)$ が互いに素のとき，式 e には一意に型がつく．

次に，型付けにもとづく well-typedness の概念について説明する．関数 f の well-typedness を保証する T-ModF1，T-ModF2 では，M で f の実装に型がつくか，f に付された契約はその型を持つかがチェックされる．ストア μ の well-typedness を保証する T-Store では μ が S と同じ定義域を持ち，互いに矛盾しないことがチェックされる．$M;\mu$ の well-typedness を保証する T-Mods では M 内のすべての関数が well-typed であり，ストアも well-typed であることがチェックされる [2]．部分関数の定義域に関する条件 $dom(\mu(m)) = dom(S(m))$ は，各モジュール m がもつ書き換え可能変数について，μ で定義されている変数の集合と，S で宣言されている変数の集合が一致することを表している条件である．

この型システムについては以下のような保存定理と進行定理が成立する．

定理 3.1 (保存) $e; M; \mu \to_m e'; M'; \mu'$，$M; \Sigma; \Gamma; S \vdash^m e : \tau$，$\Sigma; S \vdash M; \mu : \mathsf{ok}$ が成り立つとき，$M'; \Sigma; \Gamma; S \vdash^m e' : \tau$ かつ $\Sigma; S \vdash M'; \mu' : \mathsf{ok}$ が成り立つ．

[2] 本論文では基底型として nat のみを考えているが，他の基底型による拡張や，書き換え変数がより一般的な型の値を取り得る場合への対応を考慮してストアの型付けを考えている．

$b \in Base \quad ::= \quad nat$ (基底型)　$S \in MName \xrightarrow{fin} AName \xrightarrow{fin} Base$ (ストア型付け)

$\tau \in Types \quad ::= \quad b \mid \tau \to \tau$ (型)　$\Sigma \in MName \xrightarrow{fin} FName \xrightarrow{fin} Contracts$ (シグネチャ群)

Γ は組 $(x : \tau)$ の有限集合 (型環境)

$\boxed{M; \Sigma; \Gamma; S \vdash^m e : \tau}$

$$\frac{}{M; \Sigma; \Gamma; S \vdash^m \underline{n} : nat}\ (\text{T-Int}) \qquad \frac{M; \Sigma; \Gamma; S \vdash^m e_1 : nat \qquad M; \Sigma; \Gamma; S \vdash^m e_2 : nat}{M; \Sigma; \Gamma; S \vdash^m e_1\ op\ e_2 : nat}\ (\text{T-Op})$$

$$\frac{M; \Sigma; \Gamma; S \vdash^m e_1 : nat \qquad M; \Sigma; \Gamma; S \vdash^m e_2 : \tau \qquad M; \Sigma; \Gamma \cup \{x : nat\}; S \vdash^m e_3 : \tau}{M; \Sigma; \Gamma; S \vdash^m \mathsf{case}\ e_1\ \mathsf{of}\ 0 \mapsto e_2 \mid S(x) \mapsto e_3 : \tau}\ (\text{T-Case})$$

$$\frac{}{M; \Sigma; \Gamma \cup \{x : \tau\}; S \vdash^m x : \tau}\ (\text{T-Var}) \qquad \frac{(m'.f/c) \in \mathit{Intf}(M) \cup \Sigma \qquad \vdash c : \tau}{M; \Sigma; \Gamma; S \vdash^m m'.f : \tau}\ (\text{T-Fun})$$

$$\frac{M; \Sigma; \Gamma; S \vdash^m e_1 : \tau_1 \to \tau_2 \qquad M; \Sigma; \Gamma; S \vdash^m e_2 : \tau_1}{M; \Sigma; \Gamma; S \vdash^m e_1 e_2 : \tau_2}\ (\text{T-App}) \qquad \frac{M; \Sigma; \Gamma; S \vdash^m e : S(m)(a) \qquad M; \Sigma; \Gamma; S \vdash^m e' : \tau}{M; \Sigma; \Gamma; S \vdash^m (a := e; e') : \tau}\ (\text{T-Seq})$$

$$\frac{M; \Sigma; \Gamma; S \vdash^{m'} e : \tau \qquad \vdash c : \tau}{M; \Sigma; \Gamma; S \vdash^m [e]^c_{m'} : \tau}\ (\text{T-Bra}) \qquad \frac{a \in dom(S(m))}{M; \Sigma; \Gamma; S \vdash^m !a : S(m)(a)}\ (\text{T-Deref})$$

$$\frac{}{\vdash P : int}\ (\text{T-Cont1}) \qquad \frac{\vdash c_1 : \tau_1 \qquad \vdash c_2 : \tau_2}{\vdash c_1 \mapsto c_2 : \tau_1 \to \tau_2}\ (\text{T-Cont2})$$

$\boxed{\text{well-typedness}}$

$$\frac{(m.f/c = \lambda \vec{x}.e_f) \in M \qquad M; \Sigma; x_1 : \tau_1, ..., x_i : \tau_i; S \vdash^m e_f : \tau \qquad \vdash c : \tau_1 \to ... \to \tau_i \to \tau}{M; \Sigma; S \vdash^m f : \mathsf{ok}}\ (\text{T-ModF1})$$

$$\frac{(m.f/c = m'.g) \in M \qquad M; \Sigma; \epsilon; S \vdash^m m'.g : \tau \qquad \vdash c : \tau}{M; \Sigma; S \vdash^m f : \mathsf{ok}}\ (\text{T-ModF2})$$

$$\frac{\begin{array}{c} dom(\mu) = dom(S) = dom(M) \\ \text{for all } m \in dom(M), dom(\mu(m)) = dom(S(m)) \\ \text{for all } m \in dom(M) \text{ and } a \in dom(\mu(m)), M; \emptyset; \emptyset; S \vdash \mu(m)(a) : S(m)(a) \end{array}}{M; S \vdash \mu}\ (\text{T-Store})$$

$$\frac{\text{for all } m \in dom(M) \text{ and } f \in dom(M(m)), M; \Sigma; S \vdash^m f : \mathsf{ok} \qquad M; S \vdash \mu}{\Sigma; S \vdash M; \mu : \mathsf{ok}}\ (\text{T-Mods})$$

図 3. 型システム

定理 3.2 (進行) $M;\emptyset;\emptyset;S \vdash^m e:\tau$，$\emptyset;S \vdash M;\mu:\mathsf{ok}$ が成り立つとき，以下のうちいずれかひとつが成り立つ．

- $e \in \mathit{Val}$.
- ある e'，M'，μ' に対し，$e;M;\mu \to_m e';M';\mu'$.
- ある $m' \in dom(M)$ に対し，$e;M;\mu \to_m \Uparrow_{m'}$.

4 文脈等価性

本節では，われわれの言語でプログラムを書く際に利用できる独立した部品の単位として「コンポーネント」というものを考え，二つのコンポーネントがどのような状況で使われても同じように振る舞うことを意味する「文脈等価性」という概念を定義する．

まずコンポーネントを定義する．

定義 4.1 (コンポーネント) コンポーネントとは四つ組 $(M;\mu;\Sigma^{exp};\Sigma)$ である．ここで $\Sigma^{exp}:\mathit{MName} \xrightarrow{fin} \mathit{FName} \xrightarrow{fin} \mathit{Contracts}$ である．

M はコンポーネント内で定義されたすべての関数の集合，Σ^{exp} は外部に対して公開する M のインタフェース，μ は M 内の関数を実行する際に利用することができるストア，Σ はコンポーネントの使用者に実装を提供してもらう必要のある関数の集合を意図している．これらの意図は以下の整合性の条件で明確にされる．

定義 4.2 (コンポーネントの整合性) コンポーネント $(M;\mu;\Sigma^{exp};\Sigma)$ が整合的であるとは，以下の二条件をみたすことをいう．

1. $dom(M) \cap dom(\Sigma) = \emptyset$.
2. ある S に対し，$\Sigma;S \vdash M;\mu:\mathsf{ok}$.
3. $\Sigma^{exp} \subseteq \mathit{Intf}(M)$.

3. はプログラマがモジュール群 M のすべての関数を公開しなくてもよいということを示唆している．つまり，一般のモジュール付き言語のようにモジュール内部でだけ見える補助関数を使えるということである．以下では特に断りがなくとも整合的なコンポーネントのみを考える．

次に，二つのコンポーネントの結合可能性と結合について定義する．結合可能であるとは二つのコンポーネントを合わせたときに，考えている各モジュールの実装やストアの情報がすべて見えるようにできることを意味している．つまり簡約をするうえで必要になる情報はすべてわかるようにできる，ということである．

定義 4.3 (コンポーネントの結合可能性/結合) 整合的なコンポーネント $(M_1;\mu_1;\Sigma_1^{exp};\Sigma_1)$，$(M_2;\mu_2;\Sigma_2^{exp};\Sigma_2)$ に対して，

1. $(M_1;\mu_1;\Sigma_1^{exp};\Sigma_1)$ と $(M_2;\mu_2;\Sigma_2^{exp};\Sigma_2)$ が結合可能であるとは以下が成り立つことをいう．
   (a) $dom(M_1) \cap dom(M_2) = \emptyset$ かつ $dom(\Sigma_1) \cap dom(\Sigma_2) = \emptyset$.
   (b) $\Sigma_1 \subseteq \Sigma_2^{exp}$ かつ $\Sigma_2 \subseteq \Sigma_1^{exp}$.
2. $(M_1;\mu_1;\Sigma_1^{exp};\Sigma_1)$ と $(M_2;\mu_2;\Sigma_2^{exp};\Sigma_2)$ が結合可能であるとき，両者の結合はモジュール

群とストアの組であり，以下のように定義される．

$$(M_1;\mu_1;\Sigma_1^{exp};\Sigma_1)\text{⩓}(M_2;\mu_2;\Sigma_2^{exp};\Sigma_2) = (M_1 \cup M_2;\mu_1 \cup \mu_2)$$

1. について説明する．(a) は同じ名前の関数に異なる違う実装が与えられることはなく，すべての関数の実装は M_1 か M_2 のいずれかに書いてあるということを保証する．(b) は $(M_1;\mu_1;\Sigma_1^{exp};\Sigma_1)$ 側では $(M_2;\mu_2;\Sigma_2^{exp};\Sigma_2)$ 側がエキスポートしている関数のみをインポートできるということ，およびその逆を要請している．この結合可能性の概念を用いて，二つのコンポーネントの互換性を [8] に従って定義する．

定義 4.4 (互換性) 整合的なコンポーネント $(M_1;\mu_1;\Sigma_1^{exp};\Sigma_1)$，$(M_2;\mu_2;\Sigma_2^{exp};\Sigma_2)$ が互換的であることを以下で定義する*:* 任意の整合的なコンポーネント $(M;\mu;\Sigma^{exp};\Sigma)$ に対して，以下の二条件が同値である．

1. $(M;\mu;\Sigma^{exp};\Sigma)$ と $(M_1;\mu_1;\Sigma_1^{exp};\Sigma_1)$ は結合可能．
2. $(M;\mu;\Sigma^{exp};\Sigma)$ と $(M_2;\mu_2;\Sigma_2^{exp};\Sigma_2)$ は結合可能．

この定義の下では，以下の命題が成り立つ．

命題 4.1 $(M_1;\mu_1;\Sigma_1^{exp};\Sigma_1)$ と $(M_2;\mu_2;\Sigma_2^{exp};\Sigma_2)$ が互換であるとき，かつそのときに限り，$\Sigma_1^{exp} = \Sigma_2^{exp}$ かつ $\Sigma_1 = \Sigma_2$ である．

以下，互換的なコンポーネントを考える際は，Σ^{exp} と Σ ははじめから同一の記号で書くこととする．

以上をもとに，互換的なコンポーネントに対して文脈前順序を定義する．ここで，$e;M;\mu\downarrow_m$ は，$e;M;\mu \to_m^* v;M';\mu'$（$\to_m^*$ は $\to_m$ の反射推移閉包）となる値 v，モジュール群 M'，ストア μ' が存在することを表す．

定義 4.5 (文脈前順序) 互換的なコンポーネント $(M_1;\mu_1;\Sigma^{exp};\Sigma)$，$(M_2;\mu_2;\Sigma^{exp};\Sigma)$ の間の文脈前順序関係 $(M_1;\mu_1;\Sigma^{exp};\Sigma) \lesssim (M_2;\mu_2;\Sigma^{exp};\Sigma)$ を以下で定義する：$(M_1;\mu_1;\Sigma_1^{exp};\Sigma_1)$，$(M_2;\mu_2;\Sigma^{exp};\Sigma)$ と結合可能な任意のコンポーネント $(M_*;\mu_*;\Sigma_*^{exp};\Sigma_*)$ と，$M_*;\Sigma_*;\emptyset;\emptyset \vdash^{main} e:\tau$ をみたし，部分式として $[e']_m^c$ のかたちのものを含まない任意の式 e に対して，以下の二条件が成り立つ．

1. $e;(M_*;\mu_*;\Sigma_*^{exp};\Sigma_*)\text{⩓}(M_1;\mu_1;\Sigma^{exp};\Sigma)\downarrow_{main}$ ならば
   $e;(M_*;\mu_*;\Sigma_*^{exp};\Sigma_*)\text{⩓}(M_2;\mu_2;\Sigma^{exp};\Sigma)\downarrow_{main}$．
2. ある $m \in dom(M_* \cup M_i) \cup \{main\}$ に対し，
   $e;(M_*;\mu_*;\Sigma_*^{exp};\Sigma_*)\text{⩓}(M_1;\mu_1;\Sigma^{exp};\Sigma) \to_{main}^* \Uparrow_m$ ならば
   $e;(M_*;\mu_*;\Sigma_*^{exp};\Sigma_*)\text{⩓}(M_2;\mu_2;\Sigma^{exp};\Sigma) \to_{main}^* \Uparrow_m$．

ここで文脈とは「コンポーネント $(M_i;\mu_i;\Sigma^{exp};\Sigma)$ が使用される環境」を意味し，$(M_i;\mu_i;\Sigma^{exp};\Sigma)$ と結合可能なコンポーネント $(M_*;\mu_*;\Sigma_*^{exp};\Sigma_*)$ と $main$ で実行される型の付く式 e によって特定される．これらを任意に取ったときの停止性によって前順序が定義される．この文脈前順序から以下のように自然に文脈等価性が定義される．

定義 4.6 (文脈等価性) $(M_1;\mu_1;\Sigma^{exp};\Sigma)$ と $(M_2;\mu_2;\Sigma^{exp};\Sigma)$ が文脈等価であるとは，$(M_1;\mu_1;\Sigma^{exp};\Sigma) \lesssim (M_2;\mu_2;\Sigma^{exp};\Sigma)$ かつ $(M_2;\mu_2;\Sigma^{exp};\Sigma) \lesssim (M_1;\mu_1;\Sigma^{exp};\Sigma)$ が成り立つこ

とをいい，これを $(M_1;\mu_1;\Sigma^{exp};\Sigma) \simeq (M_2;\mu_2;\Sigma^{exp};\Sigma)$ で表す．

5 トレース意味論

本節では，コンポーネント $(M;\mu;\Sigma^{exp};\Sigma)$ に対し，[8, 9] に従い，トレース意味論により意味を与える．この意味論の基本的な発想は「Σ を充足する外部コンポーネントとの間で行われうる，関数呼び出しや返り値の受け渡しといったイベントの列（のうち実行の終了までを表現するものだけ）を集めたものを $(M;\mu;\Sigma^{exp};\Sigma)$ の意味と考える」というものである．これを実現するため，操作的意味論にもとづいたラベル付き状態遷移系（Labelled Transition System, LTS）を定義する．この LTS は，直観的には，モジュール群やストアを二つのコンポーネントに分割し，一方の立場に立ってみたときの（操作的意味論にもとづく）計算の見え方を表現していると言える．この直観は，実際，次章の補題 6.1 によって正当化される．また，このような定式化のもとでは，プログラムの実行の制御が自コンポーネントと外部コンポーネントの間で移り変わりながら計算が進行していくと考えられるので,「現在どちら側に制御があり，どのような契約をみたす責任を負っているか」が明示的に表現される．これは $[e]^c_m$ や E^c_m といった式，表記によって実現されている．

ラベル付き遷移系　LTS の状態はコンフィギュレーションと呼ばれ，プログラムコンフィギュレーション，環境コンフィギュレーション，ブレームコンフィギュレーションの三種がある (図 4)．プログラムコンフィギュレーションは，自コンポーネント $(M;\mu;O;I)$ で実装されているモジュール群 M に属するモジュール m か $main$ モジュールで計算が行われていることを意味するコンフィギュレーションである．e は計算されるべき式，$\mathcal{E}$ は $E^c_m : ... : E'^{c'}_{m'}$ のかたちをした，契約 c とモジュール名 m で添字付けられた評価文脈の有限列で，実行時スタックを表している（右端がスタックの底になっている）．添字の意味については後述する．$O : MName \xrightarrow{fin} FName \xrightarrow{fin} Contracts$ には外部コンポーネントにエクスポートした関数の名前とその契約の組が，$I : MName \xrightarrow{fin} FName \xrightarrow{fin} Contracts$ には外部からインポートした関数の名前とその契約の組が入っている．それぞれ $O \subseteq Intf(M)$，$I \supseteq \Sigma$ をみたしている．環境コンフィギュレーションは，I を充足する外部コンポーネント（環境）に計算の制御が移っていることを意味するコンフィギュレーションである．ブレームコンフィギュレーションは計算の結果ブレームが生じたことを示すコンフィギュレーションである．

コンフィギュレーション間の遷移はアクション（図 4）によって引き起こされる．$oact$ は自コンポーネントから外部コンポーネントへのアクションを意味し，プログラムコンフィギュレーションを環境コンフィギュレーションかブレームコンフィギュレーションに遷移させる．$\overline{\mathsf{call}}$ アクションは外部への関数呼び出しである．引数の型に応じて二種類あるが，どちらも m' が関数呼び出しの出どころを意味している．$\overline{\mathsf{return}}$ アクションは外部からの関数呼び出しに対する返り値を返すもので，どちらも m が出どころになっている．$\overline{\mathsf{blame}}$ at m については後述する．$iact$ は外部コンポーネントから自コンポーネントへのアクションを意味し，環境コンフィギュレーションをプログラムコンフィギュレーションかブレームコンフィギュレーションに遷移させる．これらのアクションは $oact$ と主客が逆になっていると考えればよい．したがって「出どころ」のモジュールは外部コンポーネントに属する．τ アクションはプログラムコンフィギュレーションをプログラムコンフィギュレーションに遷移させる．自コンポーネントと外部コンポーネントとの間のやりとりは $oact$ と $iact$（総称して $extact$ と呼ぶ）によって表現されており，τ アクションは自コンポーネント内部で完結する計算を表現している．

次に，図 5 に挙げられた LTS の遷移規則について説明する．

規則 (Tau) によれば，τ アクションは M の下での簡約にしたがって起きる．$e \neq [n]^c_{m'}, [m.f]^c_{m'}$ の条件は，このかたちの式はそれぞれ (ORet-N) と (ORet-F1) で扱いたいために付されている．

C (コンフィギュレーション)

$$
\begin{array}{lcl}
 & ::= & \Sigma \vdash e\ in\ m; \mathcal{E}; M; \mu; O; I \quad \text{(プログラムコンフィギュレーション)} \\
 & \mid & \Sigma \vdash \mathcal{E}; M; \mu; O; I \qquad\qquad \text{(環境コンフィギュレーション)} \\
 & \mid & \Uparrow_m \qquad\qquad \text{(ブレームコンフィギュレーション)} \\
oact & ::= & \overline{\mathsf{call}}\ m.f(\underline{n})\ \mathsf{from}\ m' \mid \overline{\mathsf{call}}\ m.f(m'.g) \\
 & \mid & \overline{\mathsf{return}}\ \underline{n}\ \mathsf{from}\ m \mid \overline{\mathsf{return}}\ m.g \mid \overline{\mathsf{blame}}\ \mathsf{at}\ m \\
iact & ::= & \mathsf{call}\ m.f(\underline{n})\ \mathsf{from}\ m' \mid \mathsf{call}\ m.f(m'.g) \\
 & \mid & \mathsf{return}\ \underline{n}\ \mathsf{from}\ m \mid \mathsf{return}\ m.g \mid \mathsf{blame}\ \mathsf{at}\ m \\
extact & ::= & oact \mid iact \\
act & ::= & \tau \mid extact
\end{array}
$$

図 4. コンフィギュレーションとアクションの構文

外部コンポーネントに属する関数の呼び出しでは (OCall-N) と (OCall-F) が適用される．両者の違いは操作的意味論と同様引数として渡される値が基底型であるか関数型であるかにある．まず両規則に共通する点を述べる．右辺を見ると呼び出しが行われたときの評価文脈がスタックにプッシュされていることがわかる．これは関数の呼び出しの結果の値を文脈 E で待っているという状況を表している．この際，返り値が返ってきたあと再開される計算はもとと同じ m で行われるべきで，その返り値は $m'.f$ の事後条件をみたしているべきであるが，これを (IRet) において実現するため評価文脈 E に添字としてモジュール名と契約を付記している．前提条件に現れる InnermostBracket は $m'.f$ の事前条件をみたす責任があるモジュールを特定するための関数で，引数として渡された評価文脈の最も内側にある $[\_]^c_m$ のモジュール名 m を取り出し，評価文脈が $[\_]^c_m$ を含まない時は，関数の第二引数として受け取るデフォルトのモジュール名を返す．次に規則 (OCall-N) と規則 (OCall-F) の違いを述べる． 操作的意味論とまったく同様に，関数値が引数として渡される規則 (OCall-F) では，その値にフレッシュな名前をつけてそれを外部コンポーネントに渡すようになっている．この際 O にこの生成された名前と $m'.f$ の引数契約の組が追加される．

逆に外部コンポーネントから自コンポーネントのモジュールで実装された関数が呼ばれた場合はどうなるだろうか．この状況は (ICall) の四つの規則で記述される．基本的なしくみは操作的意味論とまったく同じであるが，関数名が引数として与えられる場合 ((ICall-F1) と (ICall-F2)) はその関数名に呼ばれた関数の事前条件を付して I に記憶する．

(ICall) で自コンポーネントに属する関数が外部コンポーネントから呼ばれ，計算して値が得られればそれを外部コンポーネントに返す必要がある．これを実現するのが (ORet) である．関数名を返す場合 ((ORet-F))，その名前とみたすべき契約の組は O に記憶される．ここで (ORet-F1) と (ORet-F1) の違いについて説明する．上述のとおり (Tau) は $[m.f]^c_{m'}$ のかたちの式に対しては適用されないから，自コンポーネント側の関数が呼び出されボディを評価した結果として関数値を外部コンポーネント返す場合に適用される規則は必ず (ORet-F2) である．したがって (ORet-F1) が適用されるのは，直前に (ICall-N1) か (ICall-F1) が適用された場合だけである．

(OBlame) は自コンポーネントで内部遷移が起きている途中でその M 内のあるモジュール m' へのブレームが発生したことを表す．これを外部コンポーネントを含めたシステム全体に通知するのがアクション $\overline{\mathsf{blame}}\ \mathsf{at}\ m'$ である．(IBlame) は逆に，外部コンポーネントにおける内部遷移中にブレームが発生したことを表す．アクション $\mathsf{blame}\ \mathsf{at}\ m$ は外部からのブレーム発生の通知を意味する．

トレース集合　以上で述べた遷移規則にもとづいて，状態遷移の列が構成できる．ここから外部遷移アクションのみの列を取り出し，これを「トレース」と呼ぶ．すなわち，図 6 の規則で構成され

$\boxed{C \xrightarrow{act} C'}$

$$\frac{e; M; \mu \to_m e'; M'; \mu' \qquad e \not\equiv [\underline{n}]^c_{m'}, [m.f]^c_{m'}}{\Sigma \vdash e \ in \ m; \mathcal{E}; M; \mu; O; I \xrightarrow{\tau} \Sigma \vdash e' \ in \ m; \mathcal{E}; M'; \mu'; O; I} \ (\textsc{Tau})$$

$$\frac{(m'.f/P \mapsto c) \in I \qquad m'' = \mathrm{InnermostBracket}(E, m)}{\Sigma \vdash E[(m'.f)\underline{n}] \ in \ m; \mathcal{E}; M; \mu; O; I \xrightarrow{\overline{\mathsf{call}}\ m'.f(\underline{n})\ \mathsf{from}\ m''} \begin{cases} \Sigma \vdash (E^c_m : \mathcal{E}); M; \mu; O; I & (n \in P) \\ \Uparrow_{m''} & (n \notin P) \end{cases}} \ (\textsc{OCall-N})$$

$$\frac{(m'.f/(c_1 \mapsto c_2) \mapsto c_3) \in I \qquad m''' = \mathrm{InnermostBracket}(E, m) \qquad (g' \text{ is fresh})}{\Sigma \vdash E[(m'.f)(m''.g)] \ in \ m; \mathcal{E}; M; \mu; O; I \xrightarrow{\overline{\mathsf{call}}\ m'.f(m'''.g')} \Sigma \vdash (E^{c_3}_m : \mathcal{E}); M + (m'''.g'/c_1 \mapsto c_2 = m''.g); \mu; O + (m'''.g'/c_1 \mapsto c_2); I} \ (\textsc{OCall-F})$$

$$\frac{}{\Sigma \vdash [\underline{n}]^P_m \ in \ m; \mathcal{E}; M; \mu; O; I \xrightarrow{\overline{\mathsf{return}}\ \underline{n}\ \mathsf{from}\ m} \begin{cases} \Sigma \vdash \mathcal{E}; M; \mu; O; I & (n \in P) \\ \Uparrow_m & (n \notin P) \end{cases}} \ (\textsc{ORet-N})$$

$$\frac{(m.f/c = impl) \in M}{\Sigma \vdash m.f \ in \ m; \mathcal{E}; M; \mu; O; I \xrightarrow{\overline{\mathsf{return}}\ m.f} \Sigma \vdash \mathcal{E}; M; \mu; O + (m.f/c); I} \ (\textsc{ORet-F1})$$

$$\frac{(f' \ is \ fresh)}{\Sigma \vdash [m'.f]^c_m \ in \ m; \mathcal{E}; M; \mu; O; I \xrightarrow{\overline{\mathsf{return}}\ m.f'} \Sigma \vdash \mathcal{E}; M + (m.f'/c = m'.f); \mu; O + (m.f'/c); I} \ (\textsc{ORet-F2})$$

$$\frac{e; M; \mu \to_m \Uparrow_{m'} \qquad e \not\equiv [n]^c_{m'}}{\Sigma \vdash e \ in \ m; \mathcal{E}; M; \mu; O; I \xrightarrow{\overline{\mathsf{blame}}\ \mathsf{at}\ m'} \Uparrow_{m'}} \ (\textsc{OBlame})$$

$$\frac{(m.f/P \mapsto c) \in O \qquad (m.f/P \mapsto c = \lambda\vec{x}.e) \in M \qquad m' \in dom(I) \cup \{main\}}{\Sigma \vdash \mathcal{E}; M; \mu; O; I} \ (\textsc{ICall-N1})$$

$$\xrightarrow{\mathsf{call}\ m.f(\underline{n})\ \mathsf{from}\ m'} \begin{cases} \Sigma \vdash m.f' \ in \ m; \mathcal{E}; M + (m.f'/c = \lambda x_2 ... x_i.e[\underline{n}/x_1]); \mu; O; I \\ \qquad\qquad (n \in P, \ \vec{x} = x_1 ... x_i \ (i > 1), \ f' \text{ is fresh}) \\ \Sigma \vdash [e[\underline{n}/x]]^c_m \ in \ m; \mathcal{E}; M; \mu; O; I & (n \in P, \vec{x} = x) \\ \Uparrow_{m'} & (n \notin P) \end{cases}$$

$$\frac{(m.f/P \mapsto c) \in O \qquad (m.f/P \mapsto c = m''.g) \in M \qquad m' \in dom(I) \cup \{main\}}{\Sigma \vdash \mathcal{E}; M; \mu; O; I \xrightarrow{\mathsf{call}\ m.f(\underline{n})\ \mathsf{from}\ m'} \begin{cases} \Sigma \vdash [(m''.g)\underline{n}]^c_m \ in \ m; \mathcal{E}; M; \mu; O; I & (n \in P) \\ \Uparrow_{m'} & (n \notin P) \end{cases}} \ (\textsc{ICall-N2})$$

$$\frac{(m.f/(c_1 \mapsto c_2) \mapsto c_3) \in O \qquad (m.f/(c_1 \mapsto c_2) \mapsto c_3 = \lambda\vec{x}.e) \in M \qquad m' \in dom(I) \cup \{main\}}{\Sigma \vdash \mathcal{E}; M; \mu; O; I} \ (\textsc{ICall-F1})$$

$$\xrightarrow{\mathsf{call}\ m.f(m'.g)} \begin{cases} \Sigma \vdash m.f' \ in \ m; \mathcal{E}; M + (m.f'/c_3 = \lambda x_2 ... x_i.e[m'.g/x_1]); \mu; O; I + (m'.g/c_1 \mapsto c_2) \\ \qquad\qquad (\vec{x} = x_1 ... x_i \ (i > 1), \ f' \text{ is fresh}) \\ \Sigma \vdash [e[m'.g/x]]^{c_3}_m \ in \ m; \mathcal{E}; M; \mu; O; I' & (\vec{x} = x) \end{cases}$$

$$\frac{(m.f/(c_1 \mapsto c_2) \mapsto c_3) \in O \qquad (m.f/(c_1 \mapsto c_2) \mapsto c_3 = m''.h) \in M \qquad m' \in dom(I) \cup \{main\} \qquad (g \ is \ fresh)}{\Sigma \vdash \mathcal{E}; M; \mu; O; I \xrightarrow{\mathsf{call}\ m.f(m'.g)} \Sigma \vdash [(m''.h)(m'.g)]^{c_3}_m \ in \ m; \mathcal{E}; M; \mu; O; I + (m'.g/c_1 \mapsto c_2)} \ (\textsc{ICallF2})$$

$$\frac{m' \in dom(I) \cup \{main\}}{\Sigma \vdash E^P_m : \mathcal{E}; M; \mu; O; I \xrightarrow{\mathsf{return}\ \underline{n}\ \mathsf{from}\ m'} \begin{cases} \Sigma \vdash E[\underline{n}] \ in \ m; \mathcal{E}; M; \mu; O; I & (n \in P) \\ \Uparrow_{m'} & (n \notin P) \end{cases}} \ (\textsc{IRet-N})$$

$$\frac{m' \in dom(I) \cup \{main\} \qquad (f \ is \ fresh)}{\Sigma \vdash E^{c_1 \mapsto c_2}_m : \mathcal{E}; M; \mu; O; I \xrightarrow{\mathsf{return}\ m'.f} \Sigma \vdash E[m'.f] \ in \ m; \mathcal{E}; M; \mu; O; I + (m'.f/c_1 \mapsto c_2)} \ (\textsc{IRet-F})$$

$$\frac{m \notin dom(M)}{\Sigma \vdash \mathcal{E}; M; \mu; O; I \xrightarrow{\mathsf{blame}\ \mathsf{at}\ m'} \Uparrow_m} \ (\textsc{IBlame})$$

図 5. LTS の遷移規則

$$\frac{}{C \overset{\epsilon}{\Rightarrow} C} \qquad \frac{C \overset{s_1}{\Rightarrow} C' \overset{s_2}{\Rightarrow} C''}{C \overset{s_1 s_2}{\Longrightarrow} C''} \qquad \frac{C \overset{\tau}{\rightarrow} C'}{C \overset{\epsilon}{\Rightarrow} C'} \qquad \frac{C \xrightarrow{extact} C'}{C \overset{extact}{\Longrightarrow} C'}$$

図 6. 外部遷移の列

る s がトレースである [3]．トレースの概念にもとづいてコンポーネント $(M; \mu; \Sigma^{exp}; \Sigma)$ の意味を与えるため，まず「コンフィギュレーションの整合性」と「$main$ モジュールを含まないコンフィギュレーション」の定義を与え，それにもとづいて，トレースのうち計算の終了までを表現しているものである完全トレースを定義する．

定義 5.1 (コンフィギュレーションの整合性)

1. 環境コンフィギュレーション $\Sigma \vdash \mathcal{E}; M; \mu; O; I$ が整合的であるとは，以下が成り立つことをいう．
   - (a) $(M; \mu; O; I)$ は整合的．
   - (b) $\mathcal{E}$ 内のすべての評価文脈の下付き添字 m について，$m \in dom(O) \cup \{main\}$．
2. プログラムコンフィギュレーション $\Sigma \vdash e\ in\ m; \mathcal{E}; M; \mu; O; I$ が整合的であるとは，以下が成り立つことをいう．
   - (a) $(M; \mu; O; I)$ は整合的．
   - (b) $\mathcal{E}$ 内のすべての評価文脈の下付き添字 m' について，$m' \in dom(O) \cup \{main\}$．
   - (c) $m \in dom(O) \cup \{main\}$
   - (d) $(M; \mu; O; I)$ の整合性から存在が保証されている S に対し，$M; I; \emptyset; S \vdash^m e : \tau$．

ここで，アクションによるコンフィギュレーションの遷移は整合性を保存することがいえる．

命題 5.1 C が整合的で，ブレームコンフィギュレーションではない C' に対して $C \xrightarrow{act} C'$ ならば C' は整合的である．

次に「コンフィギュレーションが $main$ モジュールを含まないこと」を定義する．

定義 5.2 ($main$ モジュールを含まないコンフィギュレーション)

1. 環境コンフィギュレーション $\Sigma \vdash \mathcal{E}; M; \mu; O; I$ が $main$ モジュールを含まないとは，$LastModule(\mathcal{E}) \neq main$ または $\mathcal{E} = \epsilon$ が成り立つことをいう．
   ここで，関数 LastModule は，空でないスタック $E^c_m : \cdots : E'^{c'}_{m'}$ に対し，その底の評価文脈に添字付けられたモジュール名 m' を返す関数である．
2. プログラムコンフィギュレーション $\Sigma \vdash e\ in\ m; \mathcal{E}; M; \mu; O; I$ が $main$ モジュールを含まないとは，以下が成り立つことをいう．
   - (a) $LastModule(\mathcal{E}) \neq main$ または $\mathcal{E} = \epsilon$．
   - (b) $m \neq main$

以上を用いて完全トレースを定義する．

[3] τ アクションが空列として扱われていることに注意．

定義 5.3 (完全トレース) 整合的で $main$ を含まないコンフィギュレーション C，コンフィギュレーション C'，トレース s があって，$C \stackrel{s}{\Rightarrow} C'$ であるとする．このとき s が完全トレースであるとは以下のいずれかが成り立つことをいう．

1. C' は環境コンフィギュレーションであり，$|\mathcal{E}| + \#(\overline{\mathit{call}}\ in\ s) = \#(\mathit{return}\ in\ s)$.
   ここで，$|\mathcal{E}|$ は C のスタック $\mathcal{E}$ の長さを，$\#(\overline{\mathit{call}}\ in\ s)$ は s 中の $\overline{\mathit{call}}$ アクションの数を，$\#(\mathit{return}\ in\ s)$ は s 中の return アクションの数を表している
2. $C' = \Uparrow_m$.

この定義からは，以下の命題が従う．

命題 5.2 C' を環境コンフィギュレーションとし，$C \stackrel{s}{\Rightarrow} C'$ とする．s が完全トレースであることと C' のスタックが ϵ であることは同値．

コンポーネント $(M; \mu; \Sigma^{exp}; \Sigma)$ の意味は，以下のように，$(M; \mu; \Sigma^{exp}; \Sigma)$ から自然に定義される初期環境コンフィギュレーション $\Sigma \vdash \epsilon; M; \mu; \Sigma^{exp}; \Sigma$ から延びる完全トレースの集合として定義される．

定義 5.4 (トレース集合) 整合的なコンポーネント $(M; \mu; \Sigma^{exp}; \Sigma)$ の意味 $\mathit{Traces}(M; \mu; \Sigma^{exp}; \Sigma)$ を以下で定義するトレース集合によって与える．

$\mathit{Traces}(M; \mu; \Sigma^{exp}; \Sigma)$
$= \{s \mid \exists C. \Sigma \vdash \epsilon; M; \mu; \Sigma^{exp}; \Sigma \stackrel{s}{\Rightarrow} C$ かつ s は完全トレース $\}$

ここで，初期環境コンフィギュレーション $\Sigma \vdash \epsilon; M; \mu; \Sigma^{exp}; \Sigma$ は整合的で $main$ モジュールを含まないので，この定義は well-defined である．

以下では，このトレース意味論の契約つき言語の意味論としての利点について論ずる．

単一のモジュール m からなるモジュール群 M をもつ整合的で $main$ を含まないコンフィギュレーション C を考える．$C \stackrel{s}{\Rightarrow} \Uparrow_m$ のとき，s の最後のアクションは $\overline{\mathsf{call}}\ m'.f(\underline{n})$ from m，$\overline{\mathsf{return}}\ \underline{n}$ from m，$\overline{\mathsf{blame}}$ at m のいずれかになることが LTS の構成の仕方から言える．このとき，ブレーム $\Uparrow_m$ の原因となる値は，いずれの場合も m 側のプログラム実行中に得られており，「m が所有する値」だと考えることができる．したがって m は自分が所有する値についてのみその責任を問われるといえる．これは妥当な契約検査がみたすべき性質であり，こうした性質がトレースの構成法から簡単に言えることがわれわれのトレース意味論が持つ利点であると考える．

またプログラムコンフィギュレーションのプログラムは $m \neq main$ のとき一般に $[e]^c_m\ in\ m$ のかたちをしているが，m 側のモジュール群内部で進行する計算にかかわる契約はすべてチェックせず信用するというわれわれのものより「緩い」意味論を採用すれば，e の中には $[\_]^{c'}_{m'}$ を含む式は存在しないことになる．このとき $[\_]^c_m$ はモジュール m と外部のモジュール群との間の境界を意味し，ここでのみ契約検査が行われると考えることができる．これは「モジュールの境界をまたぐ値のやりとりの監視」という契約検査の直観的なイメージをうまく捉えており，この点もわれわれのトレース意味論の利点である．

6 完全抽象性

完全抽象性 (full abstraction) はプログラミング言語の意味論に求められる性質のひとつであり，その意味論が振る舞いの異なるプログラムを正確に区別できることを意味する．われわれは前節ま

でで，コンポーネントを単位として文脈等価性を定義し，トレースの概念を用いてコンポーネントの意味を与えた．したがって，われわれのトレース意味論が完全抽象であるというのは，二つの（互換的な）コンポーネントに対するトレース集合の等しさが文脈等価性の概念を正確に特徴づけることだということができる．これを文脈前順序に一般化して以下に述べる．

予想 6.1 (完全抽象性) 互換的なコンポーネント $(M_1;\mu_1;\Sigma^{exp};\Sigma)$，$(M_2;\mu_2;\Sigma^{exp};\Sigma)$ に対し，$\mathit{Traces}(M_1;\mu_1;\Sigma^{exp};\Sigma) \subseteq \mathit{Traces}(M_2;\mu_2;\Sigma^{exp};\Sigma)$ と $(M_1;\mu_1;\Sigma^{exp};\Sigma) \lesssim (M_2;\mu_2;\Sigma^{exp};\Sigma)$ は等価である．

「$\mathit{Traces}(M_1;\mu_1;\Sigma^{exp};\Sigma) \subseteq \mathit{Traces}(M_2;\mu_2;\Sigma^{exp};\Sigma)$ ならば $(M_1;\mu_1;\Sigma^{exp};\Sigma) \lesssim (M_2;\mu_2;\Sigma^{exp};\Sigma)$」を健全性,「$(M_1;\mu_1;\Sigma^{exp};\Sigma) \lesssim (M_2;\mu_2;\Sigma^{exp};\Sigma)$ ならば $\mathit{Traces}(M_1;\mu_1;\Sigma^{exp};\Sigma) \subseteq \mathit{Traces}(M_2;\mu_2;\Sigma^{exp};\Sigma)$」を完全性と呼ぶ．われわれは現在までに健全性を示している．以下では健全性に至るまでの流れを概説し，最後に完全性を示すために必要となる定義可能性 (definability) と呼ばれる性質について説明してこれを用いた完全性の証明について述べる．

健全性　前章で見たとおり LTS は操作的意味論にもとづいて定義されているから，アクションによるコンフィギュレーションの遷移と操作的意味論における簡約関係の間には何らかの対応関係があることが期待される．これを示すための準備としてプログラムコンフィギュレーションと環境コンフィギュレーションを結合し対応する操作的意味論の状態である三つ組を作る操作を定義したい．まず評価文脈のスタックの結合を定義する．

定義 6.1 (スタックの結合可能性/結合)

1. 評価文脈のスタックの組が結合可能であることを以下のように帰納的に定義する．
   - (ϵ,ϵ) は結合可能．
   - (ϵ, E^{c}_{main}) は結合可能．
   - $(\mathcal{E}_2,(E^{c}_{m}:\mathcal{E}_1))$ が結合可能なとき，
$((E^{c}_{m}:\mathcal{E}_1),(([E']^{c}_{m'})^{c'}_{m'}:\mathcal{E}_2))$ も結合可能．

   ($((E^{c}_{m}:\mathcal{E}),\epsilon)$ は結合可能でないことに注意)
2. 結合可能な評価文脈のスタックの組 $(\mathcal{E}_1,\mathcal{E}_2)$ に対し，その結合を以下のように帰納的に定義する．(結合の結果は単一の評価文脈になる)
   - $\epsilon \owedge \epsilon = []$ （$\mathcal{E}_1=\mathcal{E}_2=\epsilon$ のとき）
   - $\mathcal{E}_1 \owedge (E^{c}_{m}:\mathcal{E}'_2) = (\mathcal{E}'_2 \owedge \mathcal{E}_1)[E]$　($\mathcal{E}_2=(E^{c}_{m}:\mathcal{E}'_2)$ のとき)

この定義で重要なのは，結合可能性の定義の三つ目において E^{c}_{m} と $[E']^{c}_{m'}$ で契約 c の一致を要求している点である．この条件の意図について説明する．結合の定義を見ると，$[E']^{c}_{m'}$ の穴に何らかの式 e を入れた結果である $[E'[e]]^{c}_{m'}$ が E の穴に入れられることが意図されていることがわかる．このことと，$[E'[e]]^{c}_{m'}$ の上付き添字は $E'[e]$ の評価結果がみたすべき契約を，E^{c}_{m} の上付き添字は E の穴に入る式がみたすべき契約を意味することとを考え合わせれば，両者が一致すべきであることは自然な要求であると言えよう．

このスタックの結合と第 4 節で定義したコンポーネントの結合を用いて，以下のように整合的なコンフィギュレーション同士の結合を定義する．

定義 6.2 (コンフィギュレーションの結合可能性/結合) 整合的なプログラムコンフィギュレーションと整合的な環境コンフィギュレーションの組

$$C_1 = \Sigma_1 \vdash e\ in\ m; \mathcal{E}_1; M_1; \mu_1; O_1; I_1 \qquad C_2 = \Sigma_2 \vdash \mathcal{E}_2; M_2; \mu_2; O_2; I_2$$

に対し，

1. C_1 と C_2 が結合可能であるとは，以下が成り立つことをいう．
   (a) $(M_1; \mu_1; O_1; I_1)$ と $(M_2; \mu_2; O_2; I_2)$ は結合可能である．
   (b) $(\mathcal{E}_1, \mathcal{E}_2)$ は結合可能である．さらに，
   - $\mathcal{E}_2 = \epsilon$ のとき，$m = main$ である．
   - $\mathcal{E}_2 \neq \epsilon$ のとき，$\mathcal{E}_2$ の先頭の評価文脈を $E^c_{m'}$ とすると，ある e' に対し $e = [e']^c_m$ であるか $m.f/c = impl \in M_1$ で $e = m.f$ である．

   (c) $O_1 = I_2$ かつ $O_2 = I_1$.
2. C_1 と C_2 が結合可能であるとき，両者の結合 $C_1 \between C_2$ を以下のように定義する．

$$C_1 \between C_2 = ((\mathcal{E}_1 \between \mathcal{E}_2)[e]; (M_1; \mu_1; O_1; I_1) \between (M_2; \mu_2; O_2; I_2))$$

結合の結果は，操作的意味論の状態を表す三つ組である．

また $C_1 = \Uparrow_m$ または $C_2 = \Uparrow_m$ のときも両者は結合可能であり，その結合は $C_1 \between C_2 = \Uparrow_m$ で定義される．

以上の定義をもとに，先に述べた，アクションによるコンフィギュレーションの遷移と操作的意味論における簡約関係の間の対応関係を完全抽象性の補題として証明する．この補題により「トレースとその双対トレースのペア」と「操作的意味論にもとづく $main$ 上の簡約列」の間を行き来することが可能となる．そうした意味でこの補題はそれ自身トレース意味論の妥当性を保証するものだと言うことができる．ここで，トレース s の双対トレース $\overline{s}$ とは，トレース中の各アクションについて上線のないものに上線をつけ，上線のあるものから上線を除いて得られるトレースのことである．例えば，$s = \overline{\mathsf{call}}\ m.f(\underline{n})$ from $m' \cdot \mathsf{return}\ \underline{n}'$ from m の双対トレースは $\overline{s} = \mathsf{call}\ m.f(\underline{n})$ from $m' \cdot \overline{\mathsf{return}}\ \underline{n}'$ from m である．

補題 6.1 (Trace Composition/Decomposition) 結合可能な C_1，C_2 に対し，

1. $C_1 \overset{s}{\Rightarrow} C_1'$ かつ $C_2 \overset{\overline{s}}{\Rightarrow} C_2'$ のとき，C_1' と C_2' は結合可能であり，
   $C_1 \between C_2 \to^*_{main} C_1' \between C_2'$ が成り立つ．
2. $C_1 \between C_2 \to^*_{main} e'; M'; \mu'$（または $\Uparrow_m$） のとき，適当な C_1'，C_2'，s に対して，$C_1 \overset{s}{\Rightarrow} C_1'$，$C_2 \overset{\overline{s}}{\Rightarrow} C_2'$，$C_1'$ と C_2' は結合可能，$C_1' \between C_2' = (e'; M'; \mu')$（または $\Uparrow_m$）とできる．

Trace Decomposition に関しては，e' が値であるときについて以下の系が成り立つ．

系 6.1 補題 *6.1* の *2.* において，$e' = v$ のとき，C_1 を $main$ を含まないコンフィギュレーションとすると，s として完全トレースであるものをとることができる．

さらに，ブレームコンフィギュレーションへの遷移に関する事実として以下が成り立つことが示される．

命題 6.1 C_1，C_2 が結合可能で，$C_1 \xRightarrow{extact} \Uparrow_m$，$C_2 \xRightarrow{\overline{extact}} C_2'$ のとき，$C_2' = \Uparrow_m$ である．

以上を用いてわれわれは以下の健全性を示した．

定理 6.1 (健全性) 互換的なコンポーネント $(M_1;\mu_1;\Sigma^{exp};\Sigma)$，$(M_2;\mu_2;\Sigma^{exp};\Sigma)$ に対し，
$Traces(M_1;\mu_1;\Sigma^{exp};\Sigma) \subseteq Traces(M_2;\mu_2;\Sigma^{exp};\Sigma)$ ならば
$(M_1;\mu_1;\Sigma^{exp};\Sigma) \lesssim (M_2;\mu_2;\Sigma^{exp};\Sigma)$ である．

完全性に向けて　完全性を証明するには定義可能性 (definability) という性質を補題として使うことが一般的である ([8, 9])．定義可能性とは，直観的には「$Traces(M;\mu;\Sigma^{exp};\Sigma)$ に属する任意の完全トレース s に対し，その双対トレース $\overline{s}$ でのみ遷移できるプログラムコンフィギュレーションを作ることができる」ことを意味する．われわれは s の開始するコンフィギュレーションを一般の環境コンフィギュレーションに一般化した以下のような命題が成り立つことを予想している．

予想 6.2 (定義可能性) $C \overset{s}{\Rightarrow} C'$，$C$ は環境コンフィギュレーション，s は完全トレースであることを仮定する．このとき，ある C と結合可能なプログラムコンフィギュレーション C_* があって，任意のアクション列 t について以下が等価となる．

1. ある C_*' に対して $C_* \overset{t}{\Rightarrow} C_*'$.
2. $t \leq \overline{s}$

ここで $s_1 \leq s_2$ は s_1 が s_2 の *prefix* であることを意味する半順序関係である．

また特に C のスタックが ϵ であるとき，C_* のプログラム e として，部分式に $[e']^c_m$ のかたちのものを含まないものをとることができる．

ここで完全性の証明にとって本質的に重要なのは，C_* から延びうるトレース列は必ず $\overline{s}$ の prefix となっているということである．証明においては，与えられた s に応じてプログラムカウンタに相当するものを構成し，呼ばれるべきタイミングでのみ本体が実行される関数たちを M_* として定義できることを示す必要があると考えられる．

この定義可能性を仮定した完全性の証明を以下に示す．

命題 6.2 (定義可能性 ⇒ 完全性) 定義可能性（予想 *6.2*）が成り立つとき，互換的なコンポーネント $(M_1;\mu_1;\Sigma^{exp};\Sigma)$，$(M_2;\mu_2;\Sigma^{exp};\Sigma)$ に対し，$(M_1;\mu_1;\Sigma^{exp};\Sigma) \lesssim (M_2;\mu_2;\Sigma^{exp};\Sigma)$ ならば $Traces(M_1;\mu_1;\Sigma^{exp};\Sigma) \subseteq Traces(M_2;\mu_2;\Sigma^{exp};\Sigma)$ である．

証明　$s \in Traces(M_1;\mu_1;\Sigma^{exp};\Sigma)$ を任意にとる．s は完全トレースなので $C_1 = \Sigma \vdash \epsilon;(M_1;\mu_1;\Sigma^{exp};\Sigma)$ とすると，$C_1 \overset{s}{\Rightarrow} C_1' = \Sigma \vdash \epsilon;...$ か $C_1 \overset{s}{\Rightarrow} C_1' = \Uparrow_m$ のいずれかが成り立つ．

1. （$C_1 \overset{s}{\Rightarrow} C_1' = \Sigma \vdash \epsilon;...$ のとき）定義可能性から C_1 と結合可能な C_* があって，$C_* \overset{\overline{s}}{\Rightarrow} C_*'$ である．このとき *Trace Composotion* から，$C_1 \between C_* \rightarrow^*_{main} C_1' \between C_*'$ である．ここで C_* を適当にとれば，$C_1' \between C_*' = v;M;\mu$ とできる．したがって，$C_* = \Sigma_* \vdash e\ in\ main\ ;\epsilon;M_*;\mu_*;\Sigma;\Sigma^{exp}$ とすると，文脈前順序の定義から，
$$e;(M_*;\mu_*;\Sigma;\Sigma^{exp}) \between (M_2;\mu_2;\Sigma^{exp};\Sigma) \rightarrow^*_{main} v;M;\mu$$
が成り立つ．$C_2 = \Sigma \vdash \epsilon;(M_2;\mu_2;\Sigma^{exp};\Sigma)$ とすると，系 *6.1* から，ある C_2'，C_*'，完全トレース t に対し，$C_2 \overset{t}{\Rightarrow} C_2'$，$C_* \overset{\overline{t}}{\Rightarrow} C_*'$ である．定義可能性から $\overline{t} \leq \overline{s}$ であり，さらに t は完全トレースだから，$\overline{t} = \overline{s}$，したがって，$t = s$ であり，$s \in Traces(M_1;\mu_1;\Sigma^{exp};\Sigma)$ である．

2. （$C_1 \stackrel{s}{\Rightarrow} C_1' =\Uparrow_m$ のとき）定義可能性から C_1 と結合可能な C_* があって，$C_* \stackrel{\bar{s}}{\Rightarrow} C_*'$ である．命題 *6.1* より $C_*' =\Uparrow_m$ となる．*Trace Composition* から，$C_1 \between C_* \rightarrow^*_{main} \Uparrow_m$ であり，$C_* = \Sigma_* \vdash e \ in \ main \ ; \epsilon; M_*; \mu_*; \Sigma; \Sigma^{exp}$ とすると，文脈前順序の定義から，

$$e; (M_*; \mu_*; \Sigma; \Sigma^{exp}) \between (M_2; \mu_2; \Sigma^{exp}; \Sigma) \rightarrow^*_{main} \Uparrow_m$$

である．*Trace Decomposition* から，$C_2 \stackrel{t}{\Rightarrow} C_2'$，$C_* \stackrel{\bar{t}}{\Rightarrow} C_*'$，$C_2' \between C_*' =\Uparrow_m$ である．ここで命題 *6.1* より，$C_2' = C_*' =\Uparrow_m$ である．したがって定義可能性から，$\bar{t} = \bar{s}$ であり，それゆえ $t = s$．以上より $s \in \mathit{Traces}(M_1; \mu_1; \Sigma^{exp}; \Sigma)$ である．

□

7 関連研究

高階契約は [6] によって導入され，関数型プログラミングに契約を取り入れることができるようになった．しかし，[6] で与えられた意味論は契約を付した式の簡約関係によるものであり，高階契約それ自身の意味については深く議論されていなかった．またこの意味論上で blame が正しく割り当てられているかどうかを判定できるような枠組みも存在していなかった．こうした点を補完する研究が [6] 以降現在まで続けられてきた．[2] は契約の意味を契約を破らない値の集合と捉える意味論を提示し，[6] における契約検査が健全かつ完全であることを示した．[5] はこれとは異なり，領域理論の枠組みで契約の意味論を構築している．こうした研究を受けて，[3] は，計算体系に計算された値に責任を持つ所有者の概念を導入することで，blame 割り当ての正しさを形式的に議論する枠組みを与えている．われわれが与えたトレース意味論は文脈とのやりとりに焦点をあてており，ここに挙げた先行研究よりも「プログラムと文脈の間の値のやりとりに関する取り決め」という契約の本質がうまく表現できていると考えている．われわれの言語では値ごとに所有者が設定されているわけではないが，モジュールをその中に出現する値の所有者と考えることができる．また，われわれの言語は，基本型に対する契約はプログラミング言語で書かれていない，依存型契約を含んでない，といった点で，ここに挙げた先行研究よりも単純な体系になっている．

本論文のトレース意味論は，Java のサブセットに対して完全抽象なトレース意味論を与えた [8] に依拠するところが大きい．また本論文における，関数にフレッシュな名前を付けてやりとりするという手法は LTS を用いて高階参照をもつ言語に対する完全抽象なトレース意味論を与えた [9] にも見られる．契約をもつ言語に対し，トレース意味論を用いた議論をしている研究としては，われわれのものの他に [4] があるが，これは，トレースに対する契約を記述するための枠組みを導入した研究であり，契約検査の正しさを議論しているものではない．

トレース意味論と関連の深いトピックとして，[7, 1] を嚆矢として 90 年代後半以降発展してきた表示的意味論の一流派であるゲーム意味論がある．ゲーム意味論はプログラムと文脈との間での相互作用に着目してプログラムの意味を与えるという点でトレース意味論に似ているが，意味としてプログラムの構文から独立した数学的対象を与えるという点，部分プログラムの意味からプログラム全体の意味が決まる合成性という性質が成り立つ点がトレース意味論とは異なる．こうした関連性については [9, 11, 13] で指摘されている．さらに二つの枠組みを関連づける研究として，[10] がある．

8 おわりに

本論文でわれわれは，高階契約とモジュールの機構を持つプログラミング言語についてトレース意味論を定義した．この意味論は，コンポーネントの意味を「コンポーネント自身とコンポーネントが使われる環境との間の値のやりとり」に着目して与えるものであり,「モジュールの境界をまた

いだ値のやりとり」を監視する契約検査についての議論を簡明なものにするという点で高階契約を持つ言語に対する意味論として既存のものより優れている．また，われわれはトレース意味論と操作的意味論との整合性の根拠として Trace Compositon/Decomposition と文脈等価性に対する健全性を示した．さらに，われわれは完全抽象性が成り立つことを予想している．

今後の課題としては，完全抽象性の証明以外に

- 第 5 節末尾で述べたような契約検査についての議論を形式的に述べ直す．
- 契約自身もプログラムであるようなより実用的な言語に対して同様の手法を応用する．
- ゲーム意味論の枠組みで高階契約計算の意味論を再構築することで，契約の意味をより抽象的・一般的な視点から考察する．

といったものが考えられる．

謝辞　関数値をモジュール間で受け渡しする際，新たな名前を付けて渡すという手法は，京都大学大学院情報学研究科博士課程所属の関山太朗氏よりいただいた,「契約の情報がそのままアクションに現れてしまうと完全性が成り立たない」という指摘にもとづいて本研究に取り入れられた．ここに記して謝意を示したい．また有益なコメントをいただいた PPL の査読者にも深く感謝する。

参考文献

[1] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Information and Computation*, 163:409–470, 1996.

[2] Matthias Blume and David McAllester. Sound and complete models of contracts. *J. Funct. Program.*, 16(4-5):375–414, July 2006.

[3] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: No more scapegoating. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 215–226, New York, NY, USA, 2011. ACM.

[4] Tim Disney, Cormac Flanagan, and Jay McCarthy. Temporal higher-order contracts. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 176–188, New York, NY, USA, 2011. ACM.

[5] Robert Bruce Findler and Matthias Blume. Contracts as pairs of projections. In *Proceedings of the 8th International Conference on Functional and Logic Programming*, FLOPS'06, pages 226–241, Berlin, Heidelberg, 2006. Springer-Verlag.

[6] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 48–59, New York, NY, USA, 2002. ACM.

[7] J.M.E. Hyland and C.-H.L. Ong. On full abstraction for PCF: I, II, and III. *Information and Computation*, 163(2):285 – 408, 2000.

[8] Alan Jeffrey and Julian Rathke. Java Jr.: A fully abstract trace semantics for a core Java language. In *In ESOP, volume 3444 of LNCS*, pages 423–438. Springer-Verlag, 2005.

[9] J. Laird. A fully abstract trace semantics for general references. In *In: 34th ICALP. Volume 4596 of LNCS*. Springer, 2007.

[10] Paul Blain Levy and Sam Staton. Transition systems over games. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, pages 64:1–64:10, New York, NY, USA, 2014. ACM.

[11] Andrzej S. Murawski and Nikos Tzevelekos. Game semantics for interface middleweight Java. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 517–528, New York, NY, USA, 2014. ACM.

[12] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[13] Nicholas Wolverson. *Game semantics for an object-oriented language*. PhD thesis, The University of Edinburgh, 2008.