let 多相を備えた漸進的型付き段階的計算における実行時型検査

矢口 紘人　　亀山 幸義

静的型付き段階的計算では生成されるコードを静的に検査できる一方，生成コードの型がコード生成段階の入力に依存する場合には，その型を静的に確定できないことがある．そのため，静的型付けと動的型付けを同一言語に統合する枠組みである漸進的型付けとの組み合わせが研究されている．本研究では，コード生成器と生成コードの双方に未知型を許し，let 多相を備えた漸進的型付き段階的計算体系を，コアーションに基づいて定式化する．本論文では，表層言語と型付け規則を定義する．また，中間言語であるコアーション計算を導入し，表層言語からコアーション計算への変換と，コアーション計算の操作的意味論を定義する．コアーション計算は，実行時型検査を明示的に表現する．本体系は，コード値に対するコアーションによって必要な型検査をコード内部へ反映し，コード実行段階まで残す．また，let 多相に伴って静的には具体化先を決定できない型変数を，実行時の値に応じて具体化する動的型推論を統合する．コード合成関数とスキーマに基づくコード生成の例を通じて，これらの機構の振る舞いを示す．

1　はじめに

1.1　背景と動機

段階的計算は，プログラムの実行を複数の段階に分け，コード生成段階（stage 0）で実行されるコード生成器によって，コード実行段階（stage 1）で実行されるコードを生成する仕組みである．MetaML に代表される静的型付き段階的計算では，コードを生成する操作である bracket と，生成中のコードへ別のコードを埋め込む操作である escape に対して型付け規則を与えることで，コード生成器を実行する前に，生成されるコードの型を検査できる [9]．さらに，コード生成では，コード合成関数を異なる型で再利用できると便利である．このような再利用には，let で束縛した同じ値を，その束縛変数の利用箇所ごとに異なる型で扱える let 多相が有用である．

しかし，生成コードの型がコード生成段階の入力に依存する場合，コード生成器を静的に型付けする時点では，生成コードの型を確定できないことがある．このような依存関係を，単純な静的型システムだけで扱うことは容易ではない．そこで，段階的計算に漸進的型付けを組み合わせ，コード生成器と生成コードの双方に未知型を許すことを考える．Siek と Taha が提案した漸進的型付けは，静的型システムに未知型を加え，既知の部分を静的に検査しながら，静的には確定しない型検査を実行時へ延期する枠組みである [6]．これにより，既知の型に関する検査を静的に行いつつ，未知型に関する検査を，必要な値が得られる実行時まで延期できる．また，漸進的型付けでは，未知型と具体型の境界で実行時型検査が必要となり，その検査はコアーションなどの型変換を用いて表現される．

以上を踏まえると，コード生成器と生成コードの双方に未知型を許し，let 多相を備える段階的計算の設計には，二つの課題がある．一つ目は，コード生成段階で完了できない型検査をコード実行段階まで残す仕組みを，どのように定式化するかである．二つ目は，let 多相において静的には具体化先を決定できない型変数（未決定型変数）を，コード型や複数の実行段階の下でどのように扱うかである．本研究では，

---

* Runtime Type Checking in Gradually Typed Staged Computation with Let-Polymorphism
This is an unrefereed paper. Copyrights belong to the Author(s).
Hiroto Yaguchi, Yukiyoshi Kameyama, 筑波大学, University of Tsukuba.

漸進的型付き段階的計算体系を設計することにより，これら二つの課題に取り組む．

1.2 スキーマに基づくコード生成例

1.1 節で述べた，コード生成段階の入力に応じて生成コードの型が変わる状況は，JSON Schema に基づく JSON 処理にも見られる．

その実例として，Ajv は JSON Schema から JavaScript の検証関数を生成する実用的なバリデータ生成器である [1]．JavaScript では，文字列や専用 API を介してコード生成器を記述する場合がある．この場合，生成コードで生じた誤りをコード生成器内の原因箇所へ対応付けにくいことがある．これに対し，型付きのコード断片を組み合わせる段階的計算では，静的に判定できる型の誤りをコード生成器の型検査で検出できる．加えて，静的に判定できない型の不整合もコード生成段階またはコード実行段階で検出できる．このように型情報を利用できることは，コード生成器の試作やデバッグに有用である．

以下では，コード生成段階で完了できない型検査を生成コードへ残す必要性を示すため，この状況を簡略化した例を考える．JSON オブジェクトの構造や各フィールドの型を記述したスキーマを入力とする．このスキーマから，二種類のコードを生成する．一つは JSON 入力がスキーマに適合するかを検査するバリデータであり，もう一つは各フィールドを指定された型の値として読み出すデコーダである．スキーマは，コード生成段階の入力として与えられる．したがって，選ばれたスキーマに応じて，生成コードの処理だけでなく，生成コードが返す値の型も変わりうる．

まず，フィールド名とその読み出し時の型の一組を指定する次のスキーマを考える．

```
FieldSchema ::= IntField(Key)
              | BoolField(Key)
```

例えば，`IntField("age")` は，`"age"`フィールドの型が整数であることを表す．`BoolField("active")` は，`"active"`フィールドの型が真偽値であることを表す．このスキーマから，二つのコード生成器を考える．`genValidator` は，フィールドが指定された型の値かを検査するコードを生成する．`genDecoder` は，その値を読み出すコードを生成する．以下では，実行すると T 型の値を返すコードの型を $\langle T \rangle$ と書き，未知型を $\star$ と書く．二つのコード生成器の型は以下のようになる．

```
genValidator :  FieldSchema →
  ⟨ Json → Bool ⟩
genDecoder :  FieldSchema →
  ⟨ Json → ⋆ ⟩
```

`genValidator` が生成するコードの結果型は，整数フィールドと真偽値フィールドのどちらを入力としても `Bool` である．したがって，`genValidator` は既存の静的型付き段階的計算でも記述できる．これに対し，`genDecoder` が生成するコードの結果型は，入力のスキーマによって異なる．整数の場合は `Int` を，真偽値の場合は `Bool` を返すコードを生成する．コード生成器を型付けする時点では，コード生成段階に与えられるスキーマが分からない．そのため，既存の静的型付き段階的計算でそのまま記述することは難しいが，結果型を $\star$ とすることで対処できる．

さらに，生成されたコードを組み合わせ，デコードした値へ 1 を加えるコード生成器を考える．上で説明した bracket と escape をそれぞれ $\langle e \rangle$ と $\sim e$ で表す．その概要を以下に示す．

```
genPlusOne s =
  let d = genDecoder s in
    ⟨ λj.( ∼d j) + 1 ⟩
```

ここで，`s` はコード生成段階のスキーマ入力，`j` はコード実行段階の JSON 入力である．`genDecoder` の生成コードの結果型は $\star$ である．生成コードが返す値を整数として利用する箇所には，その値を `Int` として扱えるかを調べる実行時型検査が自動的に挿入される．この利用箇所は bracket 内部にあるため，検査は生成コードに残る．例えば，`s = IntField("age")` として生成したコードに，次の JSON オブジェクトを `j` として与える．

```
{"age": 20, "active": true}
```

このコードは整数 20 を読み出して 21 を返す．一方，`s = BoolField("active")` として生成したコードへ同じ `j` を与えると，真偽値を整数として扱えるかを

調べる検査に失敗する．型の不整合はコード実行段階で判明する一方，検査箇所の特定と挿入はコード生成器の型検査時に行われる．

1.3 研究課題とアプローチ

1.2 節の例を踏まえ，一つ目の課題は，コード生成段階で完了できない型検査をコード実行段階まで残す仕組みを，どのように定式化するかである．この例の `genPlusOne` は，`genDecoder` が生成したデコーダを利用し，その返り値に 1 を加えるコードを生成する．そのため，デコーダの返り値に対する型検査を生成コードへ残す必要があった．コード値に対する検査では，値がコードであるかをコード生成段階で調べる検査と，コード内容の値をコード実行段階で調べる検査を区別する必要がある．本研究では，変換元型と変換先型を明示するコアーションをコード型に対しても定義する．コード型のコアーションがコード値へ適用されたとき，コード内容の検査に必要なコアーションを bracket 内部へ反映し，コード生成段階で完了できない検査を生成コードへ残す．

一方，二つ目の課題は，漸進的型付けと let 多相の組み合わせによって生じる未決定型変数を，どのように具体化するかである．1.1 節で述べた let 多相では，コード合成関数を異なる型で再利用する際に，多相型の型変数を利用箇所ごとに必要な型へ具体化する．利用箇所に未知型が関わると，型変数の具体化先を静的に決められず，実行時型検査を表すコアーション中に型変数が残る場合がある．本研究では，Miyazaki らの動的型推論（Dynamic Type Inference; DTI）を導入する [4]．未決定型変数を実行時まで保持し，実行時の値に応じて具体的な型を決定する．得られた型代入は，項とコアーションへ反映する．

1.4 関連体系との比較と本研究の貢献

本研究に関連する体系として，$\lambda^{G\circ}$ と MetaGTLC がある．Yaguchi と Kameyama の $\lambda^{G\circ}$ は，コード生成器と生成コードの双方に未知型を許し，ML 風の let 多相を備える直接の先行体系である．実行時型検査は，変換先型に基づく型キャストで表される．ここで型キャストとは，値を指定された型として扱えるかどうかを検査する操作である．型キャストは，コアーションと比べて検査の合成や効率化が難しい [10]．Chen らの MetaGTLC は，漸進的に型付けされたメタ言語から静的型付き対象コードを生成する．その意味は，コアーションを持つ MetaCC への変換によって与えられる [2]．表 1 に，本体系とこれらの体系の位置付けを示す．

本体系は，生成コードに未知型を許し，実行時型検査をコアーションで表現し，let 多相と DTI を備える．$\lambda^{G\circ}$ は，生成コードに未知型を許し，let 多相を備える一方，実行時型検査を型キャストで表し，DTI は扱わない．MetaGTLC はコアーションを備えるが，生成する対象コードは静的型付きであり，let 多相と DTI は形式体系の対象外である．したがって，本体系は，生成コードに未知型を許し，let 多相を備える $\lambda^{G\circ}$ を基礎とする．実行時型検査には，MetaGTLC で用いられるコアーションに基づく表現を採用する．さらに，未決定型変数を実行時に具体化するため，DTI を導入する．

本論文の貢献は次の 3 点である．

1. コード生成器と生成コードの双方で未知型を扱うことができ，実行時型検査をコアーションによって実現する段階的計算体系を定義する．
2. 1 に対応して，コード型に対するコアーションと，コード内容に必要な検査を bracket 内部へ反映するための操作的意味論を定義する．
3. コード生成に有用な let 多相を，コアーションに基づく本体系でも扱えるようにする．さらに，$\lambda^{G\circ}$ では扱われていない未決定型変数の問題に対応するため，DTI をコアーション計算へ組み込む．これにより，コード型と複数の実行段階の下でも，未決定型変数を実行時に具体化できる．

本論文では，表層言語と型付け規則を定義する．また，表層言語からコアーション計算への変換と，コアーション計算の操作的意味論を定義し，例によってその振る舞いを示す．変換の型保存および提案するコアーション計算の型安全性の証明は，今後の課題とする．

以下，第 2 節では背景を説明し，第 3 節では提案体系の設計を概観する．第 4 節では表層言語を，第 5

表 1　本体系，$\lambda^{G\circ}$，MetaGTLC の比較

比較項目	本体系	$\lambda^{G\circ}$	MetaGTLC
生成コードに未知型を許すか？	○	○	×
コアーションに基づくか？	○	×	○
let 多相と DTI を備えるか？	○	△	−

節では表層言語からコアーション計算への変換と，コアーション計算の操作的意味論を説明する．第 6 節では例を示し，第 7 節では関連研究を説明し，第 8 節ではまとめと今後の課題を述べる．

2　背景

本節では，提案体系の説明に必要となる段階的計算，漸進的型付け，コアーション，let 多相，および動的型推論の基本概念を説明する．

2.1　段階的計算

段階的計算は，プログラムの生成と実行を複数の実行段階（stage）に分ける計算体系である [9]．コード生成器を実行する段階をコード生成段階，そこで生成されたコードを実行する段階をコード実行段階とする．コード生成段階から順に，各実行段階へ $0, 1, 2, \ldots$ と番号を付ける．項 e からコードを作る bracket を $\langle e \rangle$，生成中のコードへ別のコードを埋め込む操作である escape を $\sim e$ と書く．また，実行すると型 τ の値を返すコードの型を $\langle \tau \rangle$ と書く．コード生成段階における簡約例を以下に示す．

$$\langle 1 + \sim\langle 2 \rangle \rangle \longrightarrow \langle 1 + 2 \rangle$$

この簡約では，内側の $\langle 2 \rangle$ で作られたコードが，escape によって外側の bracket 内へ埋め込まれる．一方，結果の $1 + 2$ はコード実行段階の計算なので，コード生成段階では簡約されない．bracket 内部の項は 1 段階先の実行段階で型付けし，escape の対象には 1 段階前の実行段階におけるコード型を要求する．この型付けにより，コード生成器の型検査を通じて，生成コードが各実行段階で適切に型付けされるかを静的に確認できる．

2.2　漸進的型付け

漸進的型付けは，静的型システムに未知型を加え，静的に型が分かる部分と実行時の検査を必要とする部分を同じプログラム内に混在させる枠組みである [6]．本論文では未知型を $\star$ と書く．未知型は検査を省略する型ではなく，型に関する情報が静的には確定していないことを表す．また，型の等しさの代わりに用いる型一貫性を $\tau_1 \sim \tau_2$ と書く．$\star$ は任意の型と一貫性を持ち，関数型同士の一貫性は，構成要素の一貫性から定まる．型一貫性は推移的ではない．例えば，$\mathbf{Int} \sim \star$ と $\star \sim \mathbf{Bool}$ は成り立つ．しかし，$\mathbf{Int} \sim \mathbf{Bool}$ は成り立たない．

型が要求される箇所では，二つの型の間に型一貫性が成り立つかを静的に確認する．値を要求された型として利用できるかは，実行時に検査する．したがって，既知の型同士の不整合は静的に拒否され，未知型を介する型の不整合は実行時に検出される．

2.3　コアーション

コアーションは，互いに一貫性を持つ型の間で値を受け渡すための変換であり，漸進的型付けの実行時型検査を明示的に表現する [3] [7]．コアーションは，変換元型と変換先型に加えて，未知型との境界におけるタグの操作，関数型に沿った変換，逐次合成などの構造を保持する．未知型との境界では，値が整数値や真偽値などの基本値，関数値，コード値のいずれの形であるかを表す ground 型のタグを用いる．具体型の値を未知型として扱うときは，その値に対応する ground 型のタグを付ける．反対に，未知型の値を具体型として利用するときは，値に付いたタグが期待する ground 型と一致するかを検査する．一致しない場合は，実行時型エラー（blame）が生じる．本体系に固有のコードコアーションは第 3.2 節で概観し，第 5 節で形式化する．

2.4　let 多相と未決定型変数

以下，let $x = w$ in e における w を let 右辺と呼ぶ．ML 風の let 多相では，let で束縛する値の型に現れ，型環境には自由に現れない型変数を一般化し，型スキームを得る．例えば，型 τ に現れる型変数 X

を一般化した型スキームを $\forall X.\tau$ と書く．let 本体に現れる束縛変数の各利用箇所では，量化された型変数を，その箇所で必要な型に置き換えて具体化する．このため，同じ値を異なる型で再利用できる．

漸進的型付けと let 多相をコアーションに基づいて実現すると，let 束縛変数の型には現れない型変数が，挿入されたコアーションに残ることがある．この型変数は，各利用箇所で let 束縛変数に与えられる静的型からは具体化できないため，実行時型検査で値が得られた時点で具体型を決定する必要がある．Miyazaki らの動的型推論（Dynamic Type Inference; DTI）は，このような未決定型変数を実行時まで保持する．さらに，値に付いた ground 型のタグに基づいて型代入を生成する [4]．生成された型代入を残りの項とコアーションへ反映することで，同じ型変数に関する以後の検査でも同じ具体型を用いる．

3 提案体系の設計概観

本節では，提案体系の全体像，コードコアーション，および let 多相に伴う未決定型変数の扱いを説明する．

3.1 提案体系の全体構成

本体系は，bracket と escape による段階的計算に，漸進的型付け，コアーション，および let 多相を組み合わせたものである．本体系では，コード生成段階の項と，bracket 内部に記述する生成コードの項の双方で未知型を利用できる．コード生成器内の未知型に由来する実行時型検査は，コード生成段階で行われる．一方，生成コード内の未知型に由来する実行時型検査は生成コードに残り，コード実行段階で行われる．

本体系では，型一貫性に基づいて表層言語を型付けする．その後，未知型を介する境界に必要なコアーションを挿入し，表層言語をコアーション計算へ変換する．コアーション計算は，実行時型検査を表すコアーション式に加えて，let 多相を扱うための明示的な型抽象と型適用を持つ．コアーション計算を操作的意味論に従って評価すると，各検査は，対象の値が得られた実行段階で行われる．型の不整合を検出した場合は blame を報告する．この構成のうち，第 4 節では表層言語の型付けを，第 5 節ではコアーション計算への変換と操作的意味論を定義する．

3.2 コード値の検査とコードコアーション

通常の値に対するコアーションには，恒等コアーション，未知型との境界で用いるタグ付けとタグ検査，逐次コアーション，および関数コアーションがある．恒等コアーションは値を変換せず，逐次コアーションは二つの検査を順に適用し，関数コアーションは引数と結果のそれぞれに必要な検査を保持する．第 1.3 節で述べたように，コード値に対する検査では，コード生成段階で行うタグ検査と，コード実行段階で行う内容検査を区別する．第 2.3 節で述べた ground 型を G とする．型 G の値にタグを付けて未知型 $\star$ へ変換するコアーションを $G!$ と書く．逆に，未知型 $\star$ の値に付いたタグを検査して型 G へ変換するコアーションを $G?^l$ と書く．上付きの l は検査位置を表すラベルであり，タグが一致しなければ blame l を生じる．コード値では $G = \langle\star\rangle$ なので，タグ付けには $\langle\star\rangle!$ を，タグ検査には $\langle\star\rangle?^l$ を用いる．例えば，未知型の値をコードとして escape する場合は $\langle\star\rangle?^l$ を適用し，escape を処理する実行段階でタグを検査する．

内容検査には，τ_1 から τ_2 へのコアーション c を内容として持つコードコアーション $\langle c\rangle$ を用いる．これは，$\langle\tau_1\rangle$ から $\langle\tau_2\rangle$ への構造的な変換を表す．コードコアーション $\langle c\rangle$ を処理するときは，その内容である c を bracket 内部へ埋め込む．すなわち，bracket で囲まれたコード値に $\langle c\rangle$ を適用すると以下のように簡約される．

$$[\langle c\rangle]\langle v\rangle \longrightarrow \langle [c]v\rangle$$

この簡約によって c は生成コードの一部となり，コード実行段階まで残る．

3.3 未決定型変数の利用箇所ごとの具体化

第 2.4 節で述べた通常の let 多相では，let で束縛する値の型変数を一般化して型スキームを得る．let 束縛変数の各利用箇所では，量化された型変数をその箇所で必要な型へ具体化する．Miyazaki らの体系における動的型推論は，let 右辺を値に限る値制限の下で定式化されている．本体系も同じ制限を採用する．

本体系の変換では，let 右辺へ挿入したコアーションに，表層言語の宣言型には現れない型変数が残る場合がある．宣言型に現れる型変数は，各利用箇所で let 束縛変数に与えられる静的型に基づいて具体化する．一方，変換後の右辺だけに現れる型変数は，この静的型から具体化できないため，ν を対応させる．ν は型そのものではなく，let 束縛値の利用時に，その時点の項全体で使用されていない型変数へ置き換えるための印である．

ν から導入された型変数は，実行時の値に付いた ground 型のタグに基づく動的型推論によって具体化される．異なる利用箇所では別の型変数を用いる．一方，同一の利用箇所では，得られた型代入を評価中の項全体へ反映する．これにより，その型変数を含む型注釈とコアーションに同じ具体型を用いる．let 束縛の一般化は第 4.3 節で定義する．変換時の利用箇所ごとの具体化は第 5.3 節，型変数の実行時具体化と型代入の反映は第 5.6・5.7 節で定義する．

4 表層言語

本節では，コード生成器を記述する表層言語の構文と型付けを定義する．本言語は，ラムダ計算に bracket，escape，未知型，および値制限付き let 多相を加えたものである．実行段階を $i, j, k \in \mathbb{N}$ で表し，型付け判断を $\Gamma \vdash^i e : \tau$ と書く．これは，型環境 Γ の下で項 e が実行段階 i において型 τ を持つことを表す．第 2.1 節のコード生成段階とコード実行段階は，それぞれ実行段階 $i = 0$ と $i = 1$ に対応する．

4.1 構文，型，型スキーム

型，型スキーム，項，型環境，および実行段階付きの表層言語の値の構文を図 1 に示す．基本型 b には整数型 **Int**，真偽値型 **Bool** などが含まれる．X は型変数，$\star$ は未知型である．静的型 ρ は未知型を含まない型であり，漸進的型 τ は未知型を含み得る．型スキーム σ は型変数を全称量化した型である．また，ground 型 G と型引数 T は第 5 節のコアーション計算で用いる．ν は型ではなく，静的には決定できない型引数を表す印であり，その扱いは第 5 節で定義する．

定数 d には基本型が与えられているものとする．適用，escape，let に付く l は，後に挿入される実行時型検査が失敗した位置を表すラベルである．型環境は，変数の型スキームとその変数を利用できる実行段階の組 $(x : \sigma)^i$ からなる．let の右辺に現れる w は，値制限で許す実行段階付きの表層言語の値である．この値を w^i と書き，実行段階ごとに図 1 の下段で定める．

実行段階 0 では，定数，ラムダ抽象，および実行段階 1 の値を内容とする bracket が値である．一方，実行段階 1 以降に残る変数，適用，let などは，実行段階 0 では評価されないコードの構成要素なので値に含める．実行段階 1 の escape は実行段階 0 で処理されるため w^1 には含めず，実行段階 2 以降に残る escape だけを w^i（$i > 1$）に含める．

4.2 型一貫性と型付け

型一貫性 $\sigma_1 \sim \sigma_2$ と表層言語の型付け規則を図 2 に示す．未知型は任意の漸進的型と一貫性を持つ．関数型とコード型については，それぞれの構成要素の一貫性から型全体の一貫性を定める．型スキームについても，同じ型変数を量化した本体同士の一貫性から定める．規則 (Var) で用いる型スキームの具体化関係 $\succ$ は 4.3 節で定義する．規則 $(Brkt)$ は bracket 内部を 1 段階先の実行段階で型付けし，規則 (Esc) は 1 段階前の実行段階で得られるコードを現在のコードへ埋め込む．規則 (App) は，関数の引数の型と実際の引数の型が一貫性を持つときに関数適用に型付けをする．関数位置の型そのものが未知型である場合は，規則 $(App\star)$ によって結果型を未知型として扱う．同様に，escape の対象が未知型である場合は規則 $(Esc\star)$ を用いる．これらの規則で必要となる実行時型検査は，第 5 節の変換によってコアーションとして挿入する．

規則 $(Letp)$ では，let 右辺を同じ実行段階の表層言語の値 w_1^i に制限する．右辺から得られる型 τ' と宣言型 τ_1 には等しさではなく一貫性を要求し，変数 x は宣言型 τ_1 を一般化した型スキームで let 本体から利用する．

型と型スキーム

$$
\begin{array}{llll}
\text{基本型} & b & ::= & \mathbf{Int} \mid \mathbf{Bool} \mid \cdots \\
\text{型変数} & X & ::= & \alpha \mid \beta \mid \cdots \\
\text{原子型} & a & ::= & b \mid \star \\
\text{ground 型} & G & ::= & b \mid \star \to \star \mid \langle \star \rangle \\
\text{静的型} & \rho & ::= & b \mid \rho \to \rho \mid \langle \rho \rangle \mid X \\
\text{漸進的型} & \tau & ::= & a \mid \tau \to \tau \mid \langle \tau \rangle \mid X \\
\text{型スキーム} & \sigma & ::= & \tau \mid \forall X.\, \sigma \\
\text{型引数} & T & ::= & \rho \mid \nu
\end{array}
$$

項と型環境

$$
\begin{array}{rl}
e ::= & x \mid d \mid \lambda x : \tau\, .e \mid (e\ e)^l \mid \langle\, e \,\rangle \mid \sim^l e \\
& \mid \mathrm{let}^l\ x\ :\tau\ =\ w \ \mathrm{in}\ e \\
\Gamma ::= & \varnothing \mid \Gamma, (x : \sigma)^i
\end{array}
$$

表層言語の値

$$
\begin{array}{rl}
w^0 ::= & d \mid \lambda x : \tau\, .e \mid \langle\, w^1 \,\rangle \\
w^1 ::= & x \mid d \mid \lambda x : \tau\, .w^1 \mid \langle\, w^2 \,\rangle \mid (w^1\ w^1)^l \\
& \mid \mathrm{let}^l\ x\ :\tau\ =\ w^1 \ \mathrm{in}\ w^1 \\
w^i ::= & x \mid d \mid \lambda x : \tau\, .w^i \mid \langle\, w^{i+1} \,\rangle \mid \sim^l w^{i-1} \mid (w^i\ w^i)^l \\
& \mid \mathrm{let}^l\ x\ :\tau\ =\ w^i \ \mathrm{in}\ w^i \qquad (i > 1)
\end{array}
$$

図 1　表層言語の構文と値

型一貫性 $\boxed{\sigma_1 \sim \sigma_2}$

$$
\frac{}{\star \sim \tau} \qquad \frac{}{\tau \sim \star} \qquad \frac{}{\tau \sim \tau} \qquad \frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2'}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'} \qquad \frac{\tau_1 \sim \tau_2}{\langle\, \tau_1 \,\rangle \sim \langle\, \tau_2 \,\rangle} \qquad \frac{\sigma_1 \sim \sigma_2}{\forall X.\, \sigma_1 \sim \forall X.\, \sigma_2}
$$

型付け判断 $\boxed{\Gamma \vdash^i e\ :\ \tau}$

$$
\frac{(x : \sigma)^i \in \Gamma \qquad \sigma \succ \tau}{\Gamma \vdash^i x\ :\ \tau}\ (Var) \qquad \frac{d : b}{\Gamma \vdash^i d\ :\ b}\ (Const) \qquad \frac{\Gamma, (x : \tau_1)^i \vdash^i e\ :\ \tau_2}{\Gamma \vdash^i \lambda x : \tau_1\, .e\ :\ \tau_1 \to \tau_2}\ (Lam)
$$

$$
\frac{\Gamma \vdash^{i+1} e\ :\ \tau}{\Gamma \vdash^i \langle\, e \,\rangle\ :\ \langle\, \tau \,\rangle}\ (Brkt) \qquad \frac{\Gamma \vdash^i e_1\ :\ \tau_1 \to \tau_2 \qquad \Gamma \vdash^i e_2\ :\ \tau' \qquad \tau' \sim \tau_1}{\Gamma \vdash^i (e_1\ e_2)^l\ :\ \tau_2}\ (App)
$$

$$
\frac{\Gamma \vdash^i e_1\ :\ \star \qquad \Gamma \vdash^i e_2\ :\ \tau}{\Gamma \vdash^i (e_1\ e_2)^l\ :\ \star}\ (App\star) \qquad \frac{\Gamma \vdash^i e\ :\ \langle\, \tau \,\rangle}{\Gamma \vdash^{i+1} \sim^l e\ :\ \tau}\ (Esc) \qquad \frac{\Gamma \vdash^i e\ :\ \star}{\Gamma \vdash^{i+1} \sim^l e\ :\ \star}\ (Esc\star)
$$

$$
\frac{\Gamma \vdash^i w_1^i\ :\ \tau' \qquad \tau' \sim \tau_1 \qquad \Gamma, (x : \forall \overline{\alpha_m}.\tau_1)^i \vdash^i e_2\ :\ \tau_2 \qquad \{\overline{\alpha_m}\} \subseteq \mathbf{Free}(\tau_1) \setminus \mathbf{Free}(\Gamma)}{\Gamma \vdash^i \mathrm{let}^l\ x\ :\tau_1\ =\ w_1^i \ \mathrm{in}\ e_2\ :\ \tau_2}\ (Letp)
$$

図 2　型一貫性と表層言語の型付け規則

4.3　一般化と具体化

型スキーム σ の具体化によって型 τ が得られることを $\sigma \succ \tau$ と書き，以下で定義する．量化された型変数は静的型 $\rho_1', \ldots, \rho_n'$ で置き換える．

$$
\begin{array}{rcl}
\sigma \succ \tau & \iff & \sigma = \forall \beta_1. \cdots \forall \beta_n. \tau_0 \\
& & \wedge\ \tau = \tau_0[\beta_1 := \rho_1', \ldots, \beta_n := \rho_n']
\end{array}
$$

規則 $(Letp)$ の $\overline{\alpha_m}$ は一般化する型変数の列である．$\mathbf{Free}(A)$ を型，型スキーム，項，または型環境 A に自由に現れる型変数の集合とする．一般化する各型変数は，宣言型 τ_1 に自由に現れ，型環境 Γ には自由に現れないものから選ぶ．let 本体では x へ型スキーム $\forall \overline{\alpha_m}.\tau_1$ を与え，利用箇所ごとに規則 (Var) で静的型へ具体化する．表層言語で一般化するのは宣言型に現れる型変数である．この具体化により，同じ let 束縛値を利用箇所ごとに異なる静的型で再利用できる．コード生成における具体例は第 6 節で示す．

5　コアーション計算への変換と操作的意味論

本節では，中間言語としてコアーション計算を導入し，表層言語からコアーション計算への変換と，コアーション計算の操作的意味論を定義する．コアーション計算は，実行時型検査を明示的に表現する．本体系の特徴は，コードコアーションが内容検査をコード実行段階まで残す点と，DTI が let 多相に伴う未決定型変数を実行時に具体化する点にある．通常の代入やラムダ計算の規則は標準的なので，必要な箇所を除いて省略する．

5.1　コアーション計算

コアーション c の構文，コアーション計算の項 e' と値 v^i，および型付け規則を図 3 に示す．コアーショ

ン計算は，第 3.2 節で概観した通常のコアーションとコードコアーション $\langle c \rangle$ に加えて，型変数に対するタグ付け $X!^l$ およびタグ検査 $X?^l$ を持つ．$X!^l$ と $X?^l$ は，X の具体型が未決定のまま，X と未知型 $\star$ の間に必要な検査を保持する．コアーション c が inert であるとは，値 v^0 への適用 $[c]v^0$ が値として扱われることをいう．本体系では，タグ付け $G!$ と関数コアーションが inert なコアーションである．

コアーション計算は，コアーション適用 $[c]e'$ と blame 項 $\mathsf{blame}\ l$ を持つ．また，let 多相を扱うため，明示的な型抽象 $\Lambda\overline{\alpha}.e'$ と型適用 $x\{\overline{T}\}$ を持つ．規則 $(TApp)$ は多相変数に型引数を明示し，型スキームの本体 τ_0 に現れない量化変数には未決定を表す ν を与える．規則 $(Letp)$ は let 右辺を型抽象として保持し，規則 $(Cast)$ は型 τ_1 から τ_2 へのコアーションを項へ適用する．前述のとおり，値 v^0 への適用が値形式となる inert なコアーションはタグ付け $G!$ と関数コアーションであり，$G!$ を適用した値は ground 型のタグを保持する．実行段階 1 以降の値には，第 4.1 節の w^i と同様に，実行段階 0 では評価されない構文に加え，型適用とコアーション適用を含める．

5.2 コアーション生成

表層言語からコアーション計算への変換では，補助関数 **coerce** を使用する．**coerce** は，一貫性を持つ型 τ_1 と τ_2 から，τ_1 を τ_2 として利用するためのコアーションを生成する．その定義を図 3 の下段に示す．**ground** τ は，未知型として扱う値の構造を表す ground 型を返す．

型変数 X には ground 型を静的に決められないため，**ground** X は定義せず，専用の $X!^l$ および $X?^l$ を生成する．図 3 下段の定義は，恒等，ground 型のタグ操作，型変数，複合型の順に場合分けする．関数コアーションは引数側と結果側の検査を持つ形に生成する．コード型同士では内容のコアーションをコードコアーションで包み，その処理を 5.5 節で定義する．

5.3 表層言語からコアーション計算への変換

型付き変換判断 $\Gamma \vdash^i e : \tau \rightsquigarrow e'$ は，型環境 Γ の下で実行段階 i の表層項 e が型 τ を持ち，コアーション計算の項 e' へ変換されることを表す．主要な規則を図 4 に示す．変数，定数，ラムダ抽象，bracket，およびコード型が既知である escape は，部分項を再帰的に変換する．

規則 $(CApp)$ は実際の引数の型を関数の引数の型へ変換するコアーションを挿入する．関数位置が未知型である $(CApp\star)$ は，関数位置の項に $\star \to \star$ のタグ検査を挿入し，実際の引数を未知型へ変換するコアーションを挿入する．同様に $(CEsc\star)$ は，未知型の値に $\langle \star \rangle$ のタグ検査を挿入する．

規則 $(CLetp)$ では，表層言語の値 w_1^i をコアーション計算の値 $v_1'^i$ へ変換し，宣言型 τ_1 へ合わせるコアーションを挿入する．$\overline{\alpha_m}$ は宣言型 τ_1 に現れ，一般化の対象となる型変数を指す．各利用箇所で let 束縛変数に与えられる静的型を，規則 $(CVarPoly)$ の型引数として与える．一方，$\overline{\beta_n}$ は変換後の右辺 $v_1'^i$ だけに現れ，その静的型からは決定できないため，型引数 ν を与える．ν の実行時の扱いは 5.7 節で定義する．

5.4 通常の評価

基本簡約 $r \xrightarrow{[S]}{}^j r'$ は，実行段階 j の簡約対象 r が r' へ簡約され，型代入 S を生成することを表す．型代入を生成しない場合は $S = []$ とする．評価文脈 $E^{i \Leftarrow j}$ は，実行段階 j の穴を持つ全体として実行段階 i の項である．その構文を図 5 に示す．関数適用を左から右へ値呼びで評価し，bracket では穴の実行段階を一つ増やし，escape では一つ減らす．評価文脈は，実行段階 0 のラムダ抽象の本体には入らず，どの実行段階でも型抽象の本体には入らない．

項の 1 ステップ評価 $e \xmapsto{[S]}{}^i e'$ は次の規則で定める．

$$\frac{r \xrightarrow{[S]}{}^j r'}{E^{i \Leftarrow j}[r] \xmapsto{[S]}{}^i (E^{i \Leftarrow j}[r'])[S]}\ (E\text{-}Step)$$

$$\frac{E^{i \Leftarrow 0} \neq [\,]}{E^{i \Leftarrow 0}[\mathsf{blame}\ l] \xmapsto{[]}{}^i \mathsf{blame}\ l}\ (E\text{-}Blame)$$

生成された型代入は簡約対象だけでなく，評価文脈を含む項全体へ適用する．この点が DTI において重要であり，5.6 節で説明する．

通常の値呼びラムダ計算，escape，恒等・逐次・関

コアーション計算の項

$$c ::= \mathbf{id}\ \tau \mid G! \mid G?^l \mid c \to c \mid c;c \mid \langle\, c \,\rangle \mid X!^l \mid X?^l$$
$$e' ::= x \mid d \mid \lambda x:\tau.e' \mid e'\ e' \mid \langle\, e' \,\rangle \mid {\sim}e' \mid [c]e' \mid \mathsf{blame}\ l \mid \mathrm{let}\ x\ :\forall\overline{\alpha}.\tau\ =\ \Lambda\overline{\alpha}.e'\ \mathrm{in}\ e' \mid x\{\overline{T}\}$$

コアーション計算の値

$$v^0 ::= d \mid \lambda x:\tau.e' \mid \langle\, v^1 \,\rangle \mid [c]v^0\ (c\ \text{は inert})$$
$$v^1 ::= x \mid d \mid \lambda x:\tau.v^1 \mid \langle\, v^2 \,\rangle \mid v^1\ v^1 \mid \mathrm{let}\ x\ :\forall\overline{\alpha_m}.\tau\ =\ \Lambda\,\overline{\alpha_m}.e'\ \mathrm{in}\ v^1 \mid x\{\overline{T_m}\} \mid [c]v^1$$
$$v^i ::= x \mid d \mid \lambda x:\tau.v^i \mid \langle\, v^{i+1} \,\rangle \mid {\sim}v^{i-1} \mid v^i\ v^i \mid \mathrm{let}\ x\ :\forall\overline{\alpha_m}.\tau\ =\ \Lambda\,\overline{\alpha_m}.e'\ \mathrm{in}\ v^i \mid x\{\overline{T_m}\} \mid [c]v^i \quad (i>1)$$

コアーション型付け $\boxed{\vdash\ c\ :\tau_1 \Rightarrow \tau_2}$

$$\frac{}{\vdash\ \mathbf{id}\ \tau\ :\tau \Rightarrow \tau} \qquad \frac{}{\vdash\ G!\ :G \Rightarrow \star} \qquad \frac{}{\vdash\ G?^l\ :\star \Rightarrow G} \qquad \frac{\vdash\ c\ :\tau_1 \Rightarrow \tau_2 \qquad c \neq \mathbf{id}\ \tau}{\vdash\ \langle\, c \,\rangle\ :\langle\, \tau_1 \,\rangle \Rightarrow \langle\, \tau_2 \,\rangle} \qquad \frac{}{\vdash\ X!^l\ :X \Rightarrow \star}$$

$$\frac{}{\vdash\ X?^l\ :\star \Rightarrow X} \qquad \frac{\vdash\ c_1\ :\tau_1' \Rightarrow \tau_1 \qquad \vdash\ c_2\ :\tau_2 \Rightarrow \tau_2'}{\vdash\ c_1 \to c_2\ :\tau_1 \to \tau_2 \Rightarrow \tau_1' \to \tau_2'} \qquad \frac{\vdash\ c_1\ :\tau_1 \Rightarrow \tau_2 \qquad \vdash\ c_2\ :\tau_2 \Rightarrow \tau_3}{\vdash\ c_1;c_2\ :\tau_1 \Rightarrow \tau_3}$$

コアーション計算の型付け $\boxed{\Gamma \vdash^i e'\ :\tau}$

$$\frac{(x:\forall\overline{\alpha_m}.\tau_0)^i \in \Gamma \qquad |\overline{T_m}| = |\overline{\alpha_m}| \qquad T_m = \nu \iff \alpha_m \notin \mathbf{Free}(\tau_0) \qquad T_m = \rho_m \iff \alpha_m \in \mathbf{Free}(\tau_0)}{\Gamma \vdash^i x\{\overline{T_m}\}\ :\ \tau_0[\alpha_m := \rho_m \mid \alpha_m \in \mathbf{Free}(\tau_0)]}\ (TApp)$$

$$\frac{\Gamma \vdash^i e'\ :\ \tau_1 \qquad \vdash\ c\ :\tau_1 \Rightarrow \tau_2}{\Gamma \vdash^i [c]e'\ :\ \tau_2}\ (Cast)$$

$$\frac{\Gamma \vdash^i e_1'\ :\ \tau_1 \qquad \Gamma, (x:\forall\overline{\alpha_m}.\tau_1)^i \vdash^i e_2'\ :\ \tau_2 \qquad \{\overline{\alpha_m}\} \cap \mathbf{Free}(\Gamma) = \emptyset}{\Gamma \vdash^i \mathrm{let}\ x\ :\forall\overline{\alpha_m}.\tau_1\ =\ \Lambda\,\overline{\alpha_m}.e_1'\ \mathrm{in}\ e_2'\ :\ \tau_2}\ (Letp)$$

ground 関数

$$\mathbf{ground}\ b = b \qquad \mathbf{ground}\ \tau_1 \to \tau_2 = \star \to \star \qquad \mathbf{ground}\ \langle\, \tau \,\rangle = \langle\, \star \,\rangle$$

coerce 関数

$$\mathbf{coerce}(\tau,\tau,l) = \mathbf{id}\ \tau \quad \mathbf{coerce}(\star,G,l) = G?^l \quad \mathbf{coerce}(G,\star,l) = G! \quad \mathbf{coerce}(\star,X,l) = X?^l \quad \mathbf{coerce}(X,\star,l) = X!^l$$
$$\mathbf{coerce}(\star,\tau,l) = \mathbf{coerce}(\star,G,l); \mathbf{coerce}(G,\tau,l) \qquad (G = \mathbf{ground}\ \tau \quad \tau \neq X)$$
$$\mathbf{coerce}(\tau,\star,l) = \mathbf{coerce}(\tau,G,l); \mathbf{coerce}(G,\star,l) \qquad (G = \mathbf{ground}\ \tau \quad \tau \neq X)$$
$$\mathbf{coerce}(\tau_1 \to \tau_2, \tau_1' \to \tau_2', l) = \mathbf{coerce}(\tau_1', \tau_1, l) \to \mathbf{coerce}(\tau_2, \tau_2', l) \qquad \mathbf{coerce}(\langle\, \tau_1 \,\rangle, \langle\, \tau_2 \,\rangle, l) = \langle\, \mathbf{coerce}(\tau_1, \tau_2, l) \,\rangle$$

図 3　コアーション計算の構文，値，型付け，およびコアーション生成（抜粋）

数コアーションの基本簡約は次のとおりである．

$$(\lambda x:\tau.e)v^0 \xrightarrow{[]}{}^0 e[x := v^0] \qquad {\sim}\langle\, v^1 \,\rangle \xrightarrow{[]}{}^1 v^1$$
$$[\mathbf{id}\ \tau]v^0 \xrightarrow{[]}{}^0 v^0 \qquad [c_1;c_2]v^0 \xrightarrow{[]}{}^0 [c_2]([c_1]v^0)$$
$$([c_1 \to c_2]v_1^0)\ v_2^0 \xrightarrow{[]}{}^0 [c_2](v_1^0\ ([c_1]v_2^0))$$

$[G!]v^0$ は ground 型 G のタグを持つ値である．タグ検査 $G?^l$ は，同じタグであれば値を取り出し，異なるタグであれば blame 項を生成する．

$$[G?^l][G!]v^0 \xrightarrow{[]}{}^0 v^0$$
$$[H?^l][G!]v^0 \xrightarrow{[]}{}^0 \mathsf{blame}\ l \quad (G \neq H)$$

タグ検査の失敗によって $\mathsf{blame}\ l$ が生成されると，規則 (*E-Blame*) はその blame 項を項全体へ伝播させる．

実行段階 1 以降に残るコアーションは，その実行段階が評価対象になるまで簡約されないため，blame 項を生成しない．その実行段階でタグ検査が行われ，

変換 $\boxed{\Gamma \vdash^i e \ : \ \tau \ \rightsquigarrow \ e'}$

$$\frac{(x : \forall \overline{\alpha_m}.\tau_0)^{\,i} \in \Gamma \qquad |\overline{T'_m}| = |\overline{\alpha_m}| \qquad T'_m = \nu \iff \alpha_m \notin \mathbf{Free}(\tau_0) \qquad T'_m = \rho'_m \iff \alpha_m \in \mathbf{Free}(\tau_0) \qquad \tau = \tau_0[\alpha_m := \rho'_m \mid \alpha_m \in \mathbf{Free}(\tau_0)]}{\Gamma \vdash^i x \ : \ \tau \ \rightsquigarrow \ x\{\overline{T'_m}\}} \ (CVarPoly)$$

$$\frac{\Gamma \vdash^i e_1 \ : \ \tau_1 \to \tau_2 \ \rightsquigarrow \ e'_1 \qquad \Gamma \vdash^i e_2 \ : \ \tau' \ \rightsquigarrow \ e'_2 \qquad \tau' \sim \tau_1}{\Gamma \vdash^i (e_1 \ e_2)^l \ : \ \tau_2 \ \rightsquigarrow \ e'_1 \ ([\mathbf{coerce}(\tau', \tau_1, l)]e'_2)} \ (CApp)$$

$$\frac{\Gamma \vdash^i e_1 \ : \ \star \ \rightsquigarrow \ e'_1 \qquad \Gamma \vdash^i e_2 \ : \ \tau \ \rightsquigarrow \ e'_2}{\Gamma \vdash^i (e_1 \ e_2)^l \ : \ \star \ \rightsquigarrow \ ([\mathbf{coerce}(\star, \star \to \star, l)]e'_1) \ ([\mathbf{coerce}(\tau, \star, l)]e'_2)} \ (CApp\star)$$

$$\frac{\Gamma \vdash^i e \ : \ \star \ \rightsquigarrow \ e'}{\Gamma \vdash^{i+1} \sim^l e \ : \ \star \ \rightsquigarrow \ \sim([\mathbf{coerce}(\star, \langle \star \rangle, l)]e')} \ (CEsc\star)$$

$$\frac{\Gamma \vdash^i w_1^i \ : \ \tau' \ \rightsquigarrow \ v_1'^i \qquad \tau' \sim \tau_1 \qquad \Gamma, (x : \forall \overline{\alpha_m \ \beta_n}.\tau_1)^i \vdash^i e_2 \ : \ \tau_2 \ \rightsquigarrow \ e'_2 \qquad \{\overline{\alpha_m}\} \subseteq \mathbf{Free}(\tau_1) \setminus \mathbf{Free}(\Gamma) \qquad \{\overline{\beta_n}\} \subseteq \mathbf{Free}(v_1'^i) \setminus \mathbf{Free}(\tau_1) \setminus \mathbf{Free}(\Gamma)}{\Gamma \vdash^i \mathrm{let}^l \ x \ : \tau_1 \ = \ w_1^i \ \mathrm{in} \ e_2 \ : \ \tau_2 \ \rightsquigarrow \ \mathrm{let} \ x \ : \forall \overline{\alpha_m \ \beta_n}.\tau_1 \ = \ \Lambda \, \overline{\alpha_m \ \beta_n} \, .[\mathbf{coerce}(\tau', \tau_1, l)]v_1'^i \ \mathrm{in} \ e'_2} \ (CLetp)$$

図 4　表層言語からコアーション計算への変換規則（抜粋）

評価文脈 $\boxed{E^{i \Leftarrow j}}$

$$E^{i \Leftarrow j} ::= [\,] \quad (j = i) \mid E^{i \Leftarrow j} \ e' \mid v^i \ E^{i \Leftarrow j} \mid \lambda x : \tau \, . E^{i \Leftarrow j} \quad (i > 0) \mid \langle \, E^{i+1 \Leftarrow j} \, \rangle$$
$$\mid \sim E^{i-1 \Leftarrow j} \quad (i > 0) \mid [c] E^{i \Leftarrow j} \mid \mathrm{let} \ x \ : \forall \overline{\alpha}.\tau \ = \ \Lambda \overline{\alpha}.e' \ \mathrm{in} \ E^{i \Leftarrow j} \ (i > 0)$$

図 5　評価文脈

失敗した時点で blame 項が生成される．

5.5　コードコアーション

コードコアーションは，コード値の内容に適用すべきコアーションを保持する．その基本簡約は次の規則 (*R-code*) である．

$$\frac{}{[\langle \, c \, \rangle]\langle \, v^1 \, \rangle \overset{[]}{\longrightarrow}^0 \langle \, [c]v^1 \, \rangle} \ (R\text{-}code)$$

この規則は実行段階 0 でコード内容を検査せず，c を bracket 内部の項へ反映する．そのため，c が表す検査はコード実行段階まで残る．

例えば，実行段階 1 で $x : \star$ とし，コード値 $\langle \, x \, \rangle$ を型 $\langle \, \mathbf{Int} \, \rangle$ のコードとして利用する場合を考える．このとき挿入されるコードコアーションは次のように簡約される．

$$[\langle \, \mathbf{Int}?^l \, \rangle]\langle \, x \, \rangle \overset{[]}{\longrightarrow}^0 \langle \, [\mathbf{Int}?^l]x \, \rangle$$

右辺では，タグ検査 $\mathbf{Int}?^l$ が bracket 内部へ埋め込まれている．したがって，x を整数として利用できるかの検査は，コード実行段階まで残る．

なお，ground 型 $\langle \star \rangle$ に対するタグ付け $\langle \star \rangle!$ とタグ検査 $\langle \star \rangle?^l$ は，実行段階 0 で値がコードかを調べるタグ操作である．これは，コード内容へ c を移す構造的なコードコアーション $\langle \, c \, \rangle$ とは異なる．

5.6　動的型推論

型変数に対するタグ検査 $X?^l$ が ground 型のタグを持つ値に適用されると，次の規則によって型変数を具体化する型代入を生成する．基本型 b の場合は $X := b$ を得る．関数値とコード値の場合は内部の型がタグだけでは分からないため，その時点の項全体に現れない新しい型変数（fresh な型変数）を導入する．

$$\frac{}{[X?^l][b!]v^0 \overset{[X:=b]}{\longrightarrow}^0 v^0} \ (R\text{-}inst\text{-}b)$$

$$\frac{Y_1, Y_2 \text{ fresh}}{[X?^l][(\star \to \star)!]v^0 \overset{[X:=Y_1 \to Y_2]}{\longrightarrow}^0 [Y_1!^l \to Y_2?^l]v^0} \ (R\text{-}inst\text{-}f)$$

$$\frac{Y \text{ fresh}}{[X?^l][\langle \star \rangle!]v^0 \overset{[X:=\langle \, Y \, \rangle]}{\longrightarrow}^0 [\langle \, Y?^l \, \rangle]v^0} \ (R\text{-}inst\text{-}c)$$

型代入を $S ::= [] \mid S, X := \rho$ とし，型，型引数，コアーション，項のいずれかを A として，$A[] := A$ お

よび $A[S, X := \rho] := (A[S])[\rho/X]$ と定める．型代入は，型抽象および全称量化による型変数の束縛関係を保つように定義する．各構文に対しては通常どおり構造的に定義するが，型変数に対するタグ付けとタグ検査には次の固有な場合を持つ．

$$(X!^l)[\rho/X]_{\mathsf{co}} := \mathbf{coerce}(\rho, \star, l)$$
$$(X?^l)[\rho/X]_{\mathsf{co}} := \mathbf{coerce}(\star, \rho, l)$$

すなわち，型変数が具体化された時点で，保持していた検査を具体型に対する通常のコアーションへ作り直す．規則 (E-$Step$) がこの代入を項全体へ反映するため，同じ型変数を含む以後の型注釈とコアーションは同じ具体型を共有する．例えば，$X := \mathbf{Int}$ の後に同じ X へ真偽値が要求されると，検査は $[\mathbf{Int}?^l][\mathbf{Bool}!]v^0$ となり blame 項を生成する．また，規則 (R-$inst$-c) は型代入 $X := \langle\, Y \,\rangle$ を生成するが，Y は未決定のままである．コードコアーションが bracket 内部へ移された後，Y はコード実行段階でコード内容のタグから具体化される．

5.7 let 多相の評価

コアーション計算の let は，型抽象した右辺を本体の型適用へ代入する．

$$\mathsf{let}\ x\ : \forall\overline{\alpha}.\tau\ =\ \Lambda\overline{\alpha}.e_1\ \mathsf{in}\ e_2 \longrightarrow^0 e_2[x := \Lambda\overline{\alpha}.e_1]$$

型適用に対する項代入を次のように定める．

$$x\{T_1, \ldots, T_n\}[x := \Lambda\alpha_1.\cdots.\Lambda\alpha_n.e] := e[\alpha_1 := \widehat{T_1}, \ldots, \alpha_n := \widehat{T_n}],$$

$$\widehat{T_k} := \begin{cases} \rho_k & (T_k = \rho_k), \\ Y_k & (T_k = \nu,\ Y_k \text{ is fresh}) \end{cases} \qquad (1 \le k \le n).$$

静的型引数 ρ_k はそのまま型抽象の変数へ代入し，ν は fresh な型変数 Y_k へ置き換える．ここでも fresh な型変数は，代入を行う時点の項全体に現れないものを選ぶ．

let 束縛変数の利用箇所ごとの具体化を，関数適用のラベルを省略した次の表層言語の項で説明する．

$$\mathsf{let}\ g : \star \to \star = \lambda x : \star.(\lambda y : \alpha\, .y)\ x\ \mathsf{in}\ (\lambda u : \star\, .g\ \mathsf{true})\ (g\ 2)$$

let 右辺はラムダ値なので値制限を満たす．α は宣言型 $\star \to \star$ には現れないが，変換後の右辺では型注釈，タグ検査 $\alpha?^l$，およびタグ付け $\alpha!^l$ に現れる．そのため，規則 ($CLetp$) は変換後の右辺に現れる α を一般化し，規則 ($CVarPoly$) は g の各利用箇所に型引数 ν を与える．変換後の主要部分を次に示す．上段は型抽象した let 右辺，下段は g true と g 2 という二つの利用箇所である．

$$\Lambda\alpha.[\mathbf{id}\ \star \to \alpha!^l](\lambda x : \star.(\lambda y : \alpha\, .y)\ ([\alpha?^l]x))$$
$$(\lambda u : \star\, .g\{\nu\}\ \mathsf{true})\ (g\{\nu\}\ 2)$$

値呼び評価により，まず右側の利用箇所 $g\{\nu\}$ 2 が評価され，その ν は fresh な型変数 X となる．この利用箇所では整数のタグに基づいて $X := \mathbf{Int}$ を得る．続いて左側の利用箇所 $g\{\nu\}$ true が評価され，別の fresh な型変数 Y に対して $Y := \mathbf{Bool}$ を得る．二つの利用箇所における簡約は次のとおりである．

$$[X?^l]([\mathbf{Int}!]2) \overset{[X:=\mathbf{Int}]}{\longrightarrow^0} 2$$
$$[Y?^l]([\mathbf{Bool}!]\mathsf{true}) \overset{[Y:=\mathbf{Bool}]}{\longrightarrow^0} \mathsf{true}$$

このように，g 2 と g true という異なる利用箇所の未決定型変数は，それぞれ X と Y として独立に具体化される．一方，各利用箇所の内部では，生成された型代入が項全体へ適用されるため，同じ型変数に対する検査は同じ具体型を用いる．

6 例

本節では，第 4 節と第 5 節で定義した let 多相とコアーションが，コード生成においてどのように用いられるかを例で示す．まず，let 多相によってコード合成関数を異なる型で再利用する例を示す．次に，第 1.2 節で示したスキーマからコードを生成する例を示す．この例では，生成コードに残るコアーションが表す検査について，コード実行段階で成功する場合と blame を生じる場合を示す．

6.1 let 多相によるコード合成

let 多相がコード生成に有用である例として，二つのコード値が表す関数を合成するコード合成関数が考えられる．ここではそのような関数 `composeCode` を考える．`composeCode` を定義する表層言語の値 w_{comp} と型 `compT` は次のように書くことができる．以下では，ラベルを省略して項を書く．

$$\mathtt{compT} = \langle\, \beta \to \gamma \,\rangle \to \langle\, \alpha \to \beta \,\rangle \to \langle\, \alpha \to \gamma \,\rangle$$
$$w_{\mathrm{comp}} = \lambda f : \langle\, \beta \to \gamma \,\rangle.\, \lambda g : \langle\, \alpha \to \beta \,\rangle.\ \langle\, \lambda x : \alpha.\, (\sim f)\ ((\sim g)\ x) \,\rangle$$

composeCode f g は，$f \circ g$ を表すコードを生成する．w_{comp} は実行段階 0 のラムダ値なので，第 4.2 節の値制限を満たす．α, β, γ が型環境 Γ に自由に現れないとき，let composeCode : compT = w_{comp} in e では，let 本体の composeCode に型スキーム $\forall\alpha.\forall\beta.\forall\gamma.$compT が与えられる．

ここで，JSON を表す型を Json とし，次のコード値が与えられているとする．

incCode	$:\langle \mathbf{Int} \to \mathbf{Int} \rangle$
intDecoder	$:\langle$ Json $\to \mathbf{Int} \rangle$
notCode	$:\langle \mathbf{Bool} \to \mathbf{Bool} \rangle$
boolDecoder	$:\langle$ Json $\to \mathbf{Bool} \rangle$

let 束縛変数 composeCode の利用箇所ごとに，異なる静的型で具体化できる．

composeCode incCode intDecoder $:\langle$ Json $\to \mathbf{Int} \rangle$,
composeCode notCode boolDecoder $:\langle$ Json $\to \mathbf{Bool} \rangle$.

このように，let 多相によって，同じコード合成関数を異なるコード型で再利用できる．

6.2 バリデータとデコーダ生成

第 1.2 節で導入した，一度に一つのフィールドを指定する簡略なスキーマを用いる．

```
FieldSchema ::= IntField(Key)
              | BoolField(Key)
```

このスキーマからバリデータとデコーダを生成する擬似コードを次に示す．

```
genValidator :  FieldSchema → ⟨ Json → Bool ⟩
genValidator s = match s with
  | IntField(k) → ⟨ λj. isInt (get j k) ⟩
  | BoolField(k) → ⟨ λj. isBool (get j k) ⟩

genDecoder :  FieldSchema → ⟨ Json → ⋆ ⟩
genDecoder s = match s with
  | IntField(k) → ⟨ λj. readInt j k ⟩
  | BoolField(k) → ⟨ λj. readBool j k ⟩
```

get は指定したフィールドの値を取り出す．isInt と isBool は Bool を返し，readInt と readBool はそれぞれ Int と Bool の値を返す．

genValidator では，どちらの分岐も結果型が静的に **Bool** と決まり，型 $\langle$ Json $\to \mathbf{Bool} \rangle$ のコードを生成する．したがって，この例では未知型を用いずに二つの分岐を共通の型で扱える．これに対し，genDecoder の整数分岐と真偽値分岐が生成するコードの型は，それぞれ $\langle$ Json $\to \mathbf{Int} \rangle$ と $\langle$ Json $\to \mathbf{Bool} \rangle$ である．コード生成段階では，どちらのスキーマが選ばれるかを静的に決定できない．そのため，従来の静的型付き段階的計算で，スキーマに応じて結果型が変わるコード生成器を記述することは難しい．本体系では，二つの結果型を $\star$ でまとめ，必要な実行時型検査を生成コードへ残す．この仕組みにより，genDecoder に必要な型の振る舞いを表現できる．

$b_1 \in \{\mathbf{Int}, \mathbf{Bool}\}$ に対して $c_{b_1} = (\mathbf{id}$ Json$) \to b_1!$ とおく．各分岐を共通のコード型へ変換するコードコアーション $\langle\, c_{b_1} \,\rangle$ は，第 5.5 節の規則 (R-*code*) によって c_{b_1} を bracket 内部へ移す．その結果，デコーダが返す b_1 型の値に $\star$ のタグを付けるコアーションが生成コードへ残る．

6.3 未知型の利用と実行結果

genDecoder が生成した関数の結果へ 1 を加えるコード生成器を考える．その擬似コードを次に示す．

```
genPlusOne :  FieldSchema → ⟨ Json → Int ⟩
genPlusOne s =
  let d = genDecoder s in
    ⟨ λj. ( ∼d j) + 1 ⟩
```

genDecoder の結果型は $\star$ である一方，加算は **Int** を要求する．したがって，bracket 内部の加算箇所には，値が **Int** のタグを持つかを調べるコアーション $\mathbf{Int}?^l$ が挿入される．一方，6.2 節で説明したコードの変換により，各分岐の生成コードにはデコード結果へタグを付けるコアーションが残る．整数分岐には $\mathbf{Int}!$ が，真偽値分岐には $\mathbf{Bool}!$ が残る．生成コードを実行すると，まずデコード結果に各分岐に対応するタグが付き，その値を加算へ渡すときに $\mathbf{Int}?^l$ がタグを検査する．

第 1.2 節と同じ次の入力 $\mathtt{j}_0$ を用いる．

```
j0 = {"age": 20, "active": true}
```

整数分岐では，デコード結果に付いた **Int** のタグが加算時に求めるタグと一致するため，次のように整数 20 を取り出せる．一方，真偽値分岐では **Bool** のタグが付くため，加算時の検査に失敗して blame を生じる．

$$[\mathbf{Int}?^l]([\mathbf{Int}!]20) \xrightarrow{[]}{}^0 20$$

$$[\mathbf{Int}?^l]([\mathbf{Bool}!]\mathsf{true}) \xrightarrow{[]}{}^0 \mathsf{blame}\ l$$

二つのスキーマに対するタグ付け，タグ検査，結果を次にまとめる．

スキーマ	タグ付け	タグ検査	結果
IntField("age")	$\mathbf{Int}!$	$\mathbf{Int}?^l$	21
BoolField("active")	$\mathbf{Bool}!$	$\mathbf{Int}?^l$	blame l

この例は，型検査時に let 束縛変数の利用箇所を特

定して検査を生成コードへ挿入し，実際の型の不整合をコード実行段階で検出する振る舞いを例示している．

7 関連研究

7.1 静的型付き段階的計算

Taha と Sheard の MetaML は，静的型付き段階的計算の基礎を与えた [9]．MetaML では，コード生成器の実行前に生成コードを型検査できる．本体系は，静的型付き段階的計算の実行段階とコード型の規律を継承する．その上で，コード生成段階の入力に依存して型を確定できない部分に，未知型と実行時型検査を許す．したがって，本研究は静的型付き段階的計算を置き換えるものではなく，静的に精密な型を与えにくいコード生成を同じ枠組みで扱う拡張と位置付けられる．

7.2 漸進的型付き段階的計算

Yaguchi と Kameyama の $\lambda^{G\circ}$ は，各実行段階で未知型を利用でき，ML 風の let 多相を備える漸進的型付き段階的計算である [10]．その操作的意味論は，中間言語への変換によって与えられる．この中間言語は，実行時型検査を変換先型に基づく型キャストとして明示する．本体系は，コード生成器と生成コードの双方に未知型を許すことと let 多相を扱うことを $\lambda^{G\circ}$ から継承し，実行時型検査の表現にコアーションを用いるとともに，未決定型変数に対する動的型推論を組み込む．

Chen らの MetaGTLC は，漸進的に型付けされたメタ言語から静的型付きの対象コードを生成する体系である [2]．コード断片を埋め込む過程で型検査を行う．その中間言語 MetaCC は，通常の値に対するコアーションに加えて，コード型に対する恒等，タグ付け，タグ検査のコアーションを持つ．これらの検査はコード生成段階で行われ，コード生成が成功すれば静的に型付けされた対象コードが得られる．これに対し，本体系では生成コード自体にも未知型を許すため，内容検査をコード生成段階で完了できるとは限らない．そこで，コード値に適用された構造的なコアーションを bracket 内部へ反映し，コード実行段階まで検査を残す．コード型に対するコアーション自体は MetaCC にも存在する．本研究は，生成コードにも未知型と実行時型検査を許す設定に，この考え方を適用する．

7.3 let 多相と動的型推論

Miyazaki らは，漸進的 Hindley–Milner 型付けのための DTI を提案した [4]．let 多相への拡張では，ν と値制限を用いて，let 束縛変数の利用箇所ごとに未決定型変数を実行時に具体化する意味論を与えた．Oshima らは，DTI と空間効率のよいコアーションを統合し，型変数を含むコアーションとコアーションへの型代入を定義している [5]．本研究は，Miyazaki らによる利用箇所ごとの未決定型変数の具体化を，本体系へ組み込む．また，Oshima らを参考に，型変数を含むコアーションへの型代入を導入する．これにより，複数の実行段階の下でも，コード内容の型変数をコード実行段階で具体化できるようにする．一方，本論文ではコアーションの正規化や合成を定義しておらず，空間効率性は今後の課題である．

8 まとめと今後の課題

本研究では，コード生成器と生成コードの双方に未知型 $\star$ を許し，let 多相を備えた漸進的型付き段階的計算体系を，コアーションに基づいて定式化した．具体的には，表層言語からコアーション計算への変換と，コアーション計算の操作的意味論を定義した．コアーション計算では，内容検査をコード実行段階まで残すコードコアーションと，let 多相に伴う未決定型変数を実行時に具体化する DTI を扱う．コード合成関数の例により，let 束縛した値を異なるコード型で再利用できることを示した．また，スキーマに基づくコード生成の例により，生成コードに残した検査がコード実行段階で成功する場合と，blame を生じる場合を例示した．

今後の課題として，まず表層言語からコアーション計算への変換の型保存と，コアーション計算の型安全性の証明がある．併せて，漸進的保証に関する性質の検証も課題である [8]．また，実行時に残るコアーションの増大を抑えるための正規化・合成，生成コー

ドを実行する機構である run の導入等も今後の課題である.

参 考 文 献

[1] Ajv Contributors: Ajv JSON Schema Validator Documentation, `https://ajv.js.org/`. Accessed August 6, 2026.

[2] Chen, T., Shetty, D., Siek, J. G., Chen, C.-H., Ma, W., Venet, A., and Liu, R.: Gradual Metaprogramming, *Proceedings of the 10th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '25)*, ACM, 2025, pp. 1–14.

[3] Henglein, F.: Dynamic Typing: Syntax and Proof Theory, *Science of Computer Programming*, Vol. 22, No. 3(1994), pp. 197–230.

[4] Miyazaki, Y., Sekiyama, T., and Igarashi, A.: Dynamic Type Inference for Gradual Hindley–Milner Typing, *Proceedings of the ACM on Programming Languages*, Vol. 3, No. POPL(2019), pp. 18:1–18:29.

[5] Oshima, Y., Sekiyama, T., and Igarashi, A.: Space-Efficient Gradual Typing with Dynamic Type Inference, *Proceedings of the 42nd JSSST Annual Conference*, September 2025, pp. 6a–2–R.

[6] Siek, J. G. and Taha, W.: Gradual Typing for Functional Languages, *Proceedings of the Scheme and Functional Programming Workshop*, University of Chicago, 2006, pp. 81–92. Technical Report TR-2006-06.

[7] Siek, J. G., Thiemann, P., and Wadler, P.: Blame and Coercion: Together Again for the First Time, *Journal of Functional Programming*, Vol. 31(2021), pp. e20.

[8] Siek, J. G., Vitousek, M. M., Cimini, M., and Boyland, J. T.: Refined Criteria for Gradual Typing, *1st Summit on Advances in Programming Languages*, LIPIcs, Vol. 32, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 274–293.

[9] Taha, W. and Sheard, T.: MetaML and Multi-Stage Programming with Explicit Annotations, *Theoretical Computer Science*, Vol. 248, No. 1–2(2000), pp. 211–242.

[10] Yaguchi, H. and Kameyama, Y.: Staged Gradual Typing, *Proceedings of the 24th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '25)*, ACM, 2025, pp. 94–104.