コレオグラフィックプログラミングのための漸進的ロケーティングの提案

望月 文香，千葉 滋

本研究では，コレオグラフィックプログラミング（CP）における漸進的ロケーティングを提案する．CP では，プログラム中の値や計算に付与されたロケーションに基づいて，各エンドポイントが実行するプログラムを生成する．しかし，ロケーションを静的に決定する必要があるため，実行時に選択されたエンドポイントで値を生成する場合，選択肢ごとに同じ後続処理を記述しなければならない．漸進的ロケーティングは，静的ロケーションと動的ロケーションを組み合わせ，値の具体的な配置先を部分的に実行時まで未決定にする．これにより，異なるロケーションで生成された値を一つの変数に格納し，共通する後続処理の重複を削減できる．また，配置先が未決定の値を，演算が要求する共配置関係に基づいて具体ロケーションへ配置する．さらに，すべてのエンドポイントで整合したロケーション情報に基づいて，演算や通信の実行主体を選択するロケーションディスパッチを導入する．本稿では，漸進的ロケーティングを取り入れた ChoreoBS の言語設計を示す．

1 はじめに

分散システムでは，複数の計算主体で実行されるプログラムの間で，通信相手や通信順序の整合性を保つ必要がある．コレオグラフィックプログラミングは，分散システム全体の振る舞いを，コレオグラフィと呼ばれる単一の大域的なプログラムとして記述するプログラミングパラダイムである [8]．コンパイラは，エンドポイントプロジェクション（Endpoint Projection，EPP）により，コレオグラフィから各計算主体の局所プログラムを生成する．EPP 可能なコレオグラフィから生成された局所プログラムは，元のコレオグラフィに記述された通信を実装し，通信の不一致に起因するデッドロックが生じない [2] [8]．

多くのコレオグラフィックプログラミング言語では，値や計算に付与されたロケーションに基づいて，それらを配置または実行するエンドポイントを決定する．[4] [6]．静的なロケーション情報には，演算や通信の実行主体をコンパイル時に決定し，送受信の不一致を防止できるという利点がある．一方で，値の配置先をコンパイル時に決定する必要があるため，実行時の制御フローに応じて配置先が変化する値を柔軟に扱うことは難しい．例えば，実行時に選択したセンサ上で測定を行う場合，測定後の処理が共通であっても，その処理をセンサごとに重複して記述する必要がある．

本研究では，コレオグラフィックプログラミングのための**漸進的ロケーティング**を提案する．漸進的ロケーティングは，静的ロケーティングと動的ロケーティングを同一プログラム内で組み合わせ，値の具体的な配置先を部分的に実行時まで未決定にできる仕組みである．具体的なロケーションが静的に決まっている部分では従来の静的検査を行い，決まっていない部分では，実行時のロケーション情報に基づいて演算や通信の実行主体を決定する．これにより，異なるロケーションで生成された値を分岐後に統一的に扱い，共通する後続処理の重複を削減できる．

コレオグラフィックプログラミングに漸進的ロケーティングを導入するにあたり，本研究では二つの設計

---

* Toward Gradual Locating for Choreographic Programming

Fumika Mochizuki, Shigeru Chiba, 東京大学情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

目標を置く．第一の目標は，プログラム中のロケーション・アノテーションを段階的に動的ロケーションへ抽象化できることである．具体ロケーションの一部を動的ロケーションへ置き換えても，実行時のロケーション検査が成功する限り，プログラムの実行結果は変化しない．また，抽象化されたロケーションに関して，静的に行われていた検査は実行時検査へ移行する．この目標を実現する上で特に問題となるのが，ロケーションを省略したリテラルの扱いである．複数のエンドポイントにまたがるコレオグラフィには，リテラルの自然なデフォルト・ロケーションが存在しない．そこで本研究では，動的配置値に，具体的な配置先が未決定である *unbound* 状態と，具体ロケーションが決定された *bound* 状態を導入し，その状態をロケーション変数によって管理する．ロケーションを省略したリテラルは *unbound* な値として生成し，後続の演算が要求する共配置関係に基づいて，その具体ロケーションを決定する．これにより，リテラルから生成される値の配置先を，周囲の値の具体ロケーションに応じて決定できるため，具体ロケーションを動的ロケーションへ置き換えた場合にも，実行時検査が成功する限り，抽象化前と同じ実行結果になる．

第二の目標は，演算や通信の実行主体を静的に決定できない場合にも，各エンドポイントが整合した実行主体を選択できることである．特に通信式では，すべてのエンドポイントが，エンドポイント間通信の要否，ならびに通信を行う場合の送信元と受信先について整合した判断を行い，動的な通信主体の選択の不一致に起因するデッドロックを生じさせないことを目指す．そのため本研究では，ロケーション変数の状態と制約について，すべてのエンドポイントが整合した情報を保持する．各エンドポイントは，この情報に基づいて，自身が演算や通信を実行すべきかを実行時に判断する．本研究では，この実行時のロケーション情報に基づく実行主体の選択を，ロケーションディスパッチと呼ぶ．

本研究の貢献は以下のとおりである．

1. 静的ロケーションと動的ロケーションを同一プログラム内で組み合わせる漸進的ロケーティングを提案する．
2. 動的配置値に unbound 状態と bound 状態を導入し，制約に基づいてロケーションを決定する実行時機構を示す．
3. ロケーション情報をすべてのエンドポイントで整合させ，演算および通信の実行主体を選択するロケーションディスパッチを提案する．

以下，第 2 章では静的ロケーションに基づく ChoreoBS とその限界を示す．第 3 章では具体例を用いて漸進的ロケーティングを説明する．第 4 章では，漸進的ロケーティングの中核となる型付けと実行時機構を，型付きラムダ計算 λ_{cbs} によって形式化する．第 5 章で関連研究を議論し，第 6 章で本稿をまとめる．

2 静的ロケーションの限界

本章では，我々が開発しているコレオグラフィックプログラミング言語 ChoreoBS を用いて，静的なロケーション情報に基づくコレオグラフィックプログラミングを説明し，その限界を明らかにする．本章で扱う ChoreoBS は，後の章で導入する漸進的ロケーティングによる拡張を含まないものとする．

静的なロケーション情報は，コレオグラフィから各エンドポイントのプログラムを安全に生成するために重要である．一方，値や計算のロケーションをコンパイル時に決定する必要があるため，実行時の制御フローに応じて値の配置先が変化するプログラムでは，同一の処理をロケーションごとに重複して記述しなければならない．本章では，複数の温度センサを用いる IoT アプリケーションを例として，この問題を示す．

2.1 コレオグラフィックプログラミング

コレオグラフィックプログラミングは，分散システム全体の振る舞いを，**コレオグラフィ**と呼ばれる単一のプログラムとして記述するプログラミングパラダイムである．コレオグラフィに参加する個々の計算主体を**エンドポイント**と呼ぶ．コレオグラフィコンパイラは，大域的に記述されたコレオグラフィから，各エンドポイントで実行される**局所プログラム**を生成する．この変換過程をエンドポイントプロジェクション（EPP）と呼ぶ．EPP は，各計算をその実行を担当するエンドポイントへ射影し，大域的に記述された通

```
function calibrateAtSensor1(
  raw: float@sensor1,
  bias: float@sensor1
): float@sensor1 {
  return raw + bias;
}

let raw: float@sensor1 =
  readTemperatureAtSensor1();
let bias: float@sensor1 = 0.5@sensor1;

let calibrated: float@sensor1 =
  calibrateAtSensor1(raw, bias);

let result: float@server <- calibrated;
storeTemperature(result);
```

リスト 1　単一センサを用いた温度測定

信を送信側と受信側の局所的な通信操作へ変換する．EPP 可能なコレオグラフィでは，同一の通信から対応する送信操作と受信操作が生成されるため，送受信の不一致に起因するデッドロックが生じない [2] [8]．

リスト 1 は，温度センサ sensor1 から測定値を取得し，その値を補正した後にサーバへ送信して保存するコレオグラフィを ChoreoBS で記述したものである．ChoreoBS では，値の配置先を**ロケーション**によって表す．値の型は，値の種類を表すベース型に，@とロケーションからなるロケーション・アノテーションを付与した形で記述される．例えば，float@sensor1 は，ベース型が float であり，sensor1 に配置された浮動小数点数の型を表す．また，リテラルにもロケーション・アノテーションを付与し，0.5@sensor1 は sensor1 に配置された浮動小数点数のリテラルを表す．

リスト 1 では，変数 raw，bias，および calibrated が sensor1 に配置されている．raw には，sensor1 で実行されるローカル関数 readTemperatureAtSensor1 が返す温度が格納される．関数 calibrateAtSensor1 は，sensor1 に配置された raw と bias を受け取り，両者の和を返す．したがって，この関数の加算も sensor1 で実行される．

ChoreoBS では，ある値の payload から，指定されたロケーションに配置される新しい値を生成する処理を，通信演算子<-によって記述する．一般に，通信式は次のように記述される．

```
let y:T@q <- e;
```

この通信式は，式 e をその配置先で評価し，得られた値をエンドポイント q へコピーして，q 上の変数 y に束縛することを表す．コピー先がコピー元の payload を保持していない場合にはエンドポイント間通信を行うが，すでに保持している場合にはローカルコピーを行う．したがって，通信式は必ずしもエンドポイント間通信を発生させるとは限らない．ChoreoBS の具体ロケーションには，sensor1 のように一つのエンドポイントを表すロケーションと，すべてのエンドポイントを表す every がある．コピー先には，これらの具体ロケーションのいずれも指定できる．コピー先が every である場合，式の評価結果はコレオグラフィに参加するすべてのエンドポイントへコピーされ，各エンドポイントにおける変数に束縛される．リスト 1 では，calibrated の値が sensor1 から server へ送信され，server 上の変数 result に束縛される．その後，storeTemperature(result) によって補正後の温度が server 上に保存される．

リスト 1 に EPP を適用すると，温度の取得と補正は sensor1 用の局所プログラムへ，温度の保存は server 用の局所プログラムへ射影される．また，通信式は，sensor1 側の送信操作 send(server, calibrated) と，server 側の受信操作 receive(sensor1) へ変換される．これらの操作は同一の通信式から生成されるため，局所プログラムを個別に記述した場合に生じ得る送受信の不一致を防ぐことができる．

2.2　実行時の配置選択とコード重複

上で示したように，一般にコレオグラフィックプログラミングでは，全ての値や計算のロケーションが静的に決まることを前提としている．しかしながら，静的ロケーションだけでは実行時の制御フローに応じて値の配置先が変化するプログラムを，共通処理の複製なしに記述することは難しい．

この問題を示すため，リスト 2 では，リスト 1 のアプリケーションにもう一つの温度センサ sensor2 を追加する．拡張後のアプリケーションは，server で取得した現在時刻に基づいて使用するセンサを決

```
let currentTime: integer@every <-
  getCurrentTimeAtServer();
let useSensor1: boolean@every =
  isMorning(currentTime);

if (useSensor1) {
  let raw1: float@sensor1 =
    readTemperatureAtSensor1();

  let bias1: float@sensor1 =
    0.5@sensor1;

  let calibrated1: float@sensor1 =
    calibrateAtSensor1(raw1, bias1);

  let result1: float@server <- calibrated1;
  storeTemperature(result1);
} else {
  let raw2: float@sensor2 =
    readTemperatureAtSensor2();

  let bias2: float@sensor2 =
    0.5@sensor2;

  let calibrated2: float@sensor2 =
    calibrateAtSensor2(raw2, bias2);

  let result2: float@server <- calibrated2;
    storeTemperature(result2);
}
```

リスト 2　実行時に使用するセンサを選択する温度測定

定し，午前中には sensor1 から，午後には sensor2 から温度を取得する．どちらのセンサを使用した場合も，取得した温度に同じ補正を適用し，補正後の値を server へ送信して保存する．

getCurrentTimeAtServer は，server で現在時刻を取得するローカル関数である．リスト 2 では，この関数が返す時刻を通信式によってすべてのエンドポイントで共有し，@every に配置された変数 currentTime に束縛する．各エンドポイントは，同じ currentTime に対して isMorning を評価するため，変数 useSensor1 もすべてのエンドポイントで同じ値を持つ．コレオグラフィの条件分岐では，関係するエンドポイントがどちらの分岐を実行するかについて整合した知識を持つ必要がある．この要件は **Knowledge of Choice** と呼ばれる．ChoreoBS では，条件式を@every に配置しなければならないとすることで，すべてのエンドポイントが同じ分岐を選択することを保証する．この設計は，複数のエンドポイントに配置された値を用いて Knowledge of Choice を満たす先行研究の方針に従う [1].

リスト 2 では，then 分岐の測定値と補正値が sensor1 に配置されるのに対し，else 分岐ではそれらが sensor2 に配置される．そのため，それぞれの分岐では，異なるロケーションを持つ補正関数 calibrateAtSensor1 と calibrateAtSensor2 が必要となる．これらの関数は引数と戻り値のロケーションだけが異なり，いずれも測定値と補正値の和を返す．さらに，温度の取得だけでなく，補正値の生成，補正計算，サーバへの通信，および保存処理も分岐ごとに記述されている．

この重複は，補正処理がセンサ固有だから生じるのではない．静的ロケーションに基づく型システムでは，sensor1 と sensor2 から得られた値を分岐後に一つの変数として扱えないため，本来は共通である後続処理まで分岐内に記述する必要がある．

省略されたロケーション・アノテーションをプログラムの静的な制約から推論することで，注釈の記述量を削減することは考えられる．しかし，推論によって得られるロケーションはコンパイル時に定まるため，実行時の制御フローに応じて変化する配置先を表すことはできない．また，ロケーション多相を用いれば，具体ロケーションを変数として抽象化することで，ロケーションだけが異なる二つの補正関数を一つの関数へ一般化できる [5]．一方，リスト 2 で必要となるのは，異なるロケーションで生成された値を分岐後に一つの変数へ合流させ，その配置先を実行時情報として後続の計算へ伝播することである．ロケーションの静的な推論やロケーション多相だけでは，このような動的な配置値の合流を直接には表現できない．

3　例で見る漸進的ロケーティング

本研究では，コレオグラフィックプログラミングのための漸進的ロケーティングを提案する．漸進的ロケーティングとは，静的ロケーティングと動的ロケーティングを同一プログラム内で組み合わせ，値の配置先を部分的に実行時に決定できるようにする仕組みである．これにより，具体ロケーションが指定された部分では静的な検査による安全性を維持しつつ，実行時に変化する値の配置先を柔軟に扱い，共通処理の重

```
function calibrate(
  raw: float@dyn,
  bias: float@dyn
): float@dyn {
  return raw + bias;
}

let currentTime: integer@every <-
  getCurrentTimeAtServer();
let useSensor1: boolean@every =
  isMorning(currentTime);

let raw: float@dyn;

if (useSensor1) {
  raw = readTemperatureAtSensor1();
} else {
  raw = readTemperatureAtSensor2();
}

let bias: float@dyn = 0.5@dyn;
let calibrated: float@dyn =
  calibrate(raw, bias);

let result: float@server <- calibrated;
storeTemperature(result);
```

リスト 3　漸進的ロケーティングを用いた温度測定

複を削減できる．本章では，漸進的ロケーティングを取り入れた ChoreoBS を用いて，動的ロケーションによるプログラムの記述方法，動的配置値の実行時表現，およびエンドポイントプロジェクションを説明する．

3.1　動的ロケーティング

動的ロケーティングは，値の具体的な配置先を静的には固定せず，プログラムの実行中に決定できるようにする仕組みである．動的ロケーティングでは，具体的な配置先が静的には決定されない値を**動的配置値**と呼び，ChoreoBS では，その型やリテラルに動的ロケーション・アノテーション`@dyn` を付与することで表す．ChoreoBS では，省略されたロケーション・アノテーションは`@dyn` として補完される．

漸進的ロケーティングでは，プログラムの各部分に対して動的ロケーションまたは具体ロケーションを選択でき，必要に応じてロケーション情報を段階的に抽象化できる．具体ロケーションを持つ値は，ベース型が同じであれば，動的ロケーションを持つ変数に代入でき，その具体ロケーションは実行時情報として保持される．逆に，動的ロケーションを持つ値も，ベース型が同じであれば，具体ロケーションを持つ変数に代入できる．ただし，実行時のロケーションが代入先の具体ロケーションと整合しない場合には，ランタイムエラーとなる．

リスト 3 は，リスト 2 のプログラムを，漸進的ロケーティングを取り入れた ChoreoBS で書き直したものである．リスト 3 では，動的ロケーションを明示するために`@dyn` を記述しているが，これらのロケーション・アノテーションはすべて省略できる．リスト 2 ではセンサごとに記述していた補正処理とサーバへの通信を，リスト 3 では条件分岐の外側に移動して共通化している．条件分岐内には，使用するセンサから測定値を取得する処理だけが残されている．

変数 `raw` と `bias`，関数 `calibrate` の仮引数 `raw` と `bias`，およびその戻り値は，いずれも動的ロケーションを持つ．したがって，これらにはベース型が一致する任意のロケーションを持つ値を代入できる．`readTemperatureAtSensor1()` は `sensor1` に配置された値を返し，その値を動的ロケーションを持つ変数 `raw` に代入する．同様に，`readTemperatureAtSensor2()` が返す `sensor2` の値も `raw` に代入できる．このため，実行時に選択されたセンサが異なっても，分岐後には同じ変数 `raw` を使用できる．

リテラル `0.5@dyn` は，動的配置値を生成し，動的ロケーションを持つ変数 `bias` に代入される．関数 `calibrate` の呼び出しでは，変数 `raw` と `bias` に格納された動的配置値が，それぞれ動的ロケーションを持つ仮引数に渡される．関数内の加算は，これらの値の実行時のロケーションに基づいて実行され，その結果も動的配置値として返される．動的配置値の実行時のロケーションがどのように決定されるかについては，次節で説明する．

3.2　動的配置値の実行時状態

動的配置値は，値の本体を表す payload と，その論理的な配置先を表すロケーション変数からなる．ロケーション変数はそれ自体では具体ロケーションを表さず，その配置先は，ロケーション変数の間に追加

される制約によって定まる．動的配置値は，具体ロケーションが未決定である *unbound* 状態と，具体ロケーションが決定された *bound* 状態のいずれかを取る．ロケーション変数に対する制約から具体ロケーションが定まらない場合，動的配置値は *unbound* であり，制約から `sensor1` や `every` などの具体ロケーションが定まる場合，その値はそのロケーションに *bound* されている．

unbound 状態は，具体ロケーションが指定されていないリテラルの初期状態を表すために導入する．単一のエンドポイント上で実行されるプログラムでは，リテラルはそのエンドポイントで評価され，生成された値も同じエンドポイントに配置される．一方，コレオグラフィでは，ロケーションが指定されていないリテラルをどのエンドポイントで評価するかが定まらない．そのため，`@dyn` が付与されたリテラルを初めから特定のエンドポイントに配置するのではなく，その具体ロケーションを unbound として表す．

具体ロケーションを持つ値から生成された動的配置値は，初めから元の値のロケーションに bound されている．一方，`@dyn` が付与されたリテラルは，新たに生成されたロケーション変数を持つ unbound な動的配置値を生成する．unbound とは値が未計算であることではなく，その値を配置する具体ロケーションが未決定であることを表す．

unbound な動的配置値は，実行時の表現としてすべてのエンドポイントに payload を持つが，論理的にすべてのエンドポイントへ配置された値ではない．この点で，unbound な値は@every が付与された値と異なる．@every が付与された値は，すべてのエンドポイントに配置され，各エンドポイントから利用できることが保証された値である．これに対して unbound な動的配置値は，具体ロケーションが未決定であり，後にいずれかの具体ロケーションへ bound され得る値である．

加算など，複数の値が同じロケーションに配置されていることを要求する演算では，それらの値のロケーション変数が等しいという制約が追加される．両方の値が unbound である場合，具体ロケーションは未決定のままであるが，一方のロケーションが決定されると，追加された制約によって他方も同じロケーションに決定される．一方の値がすでに具体ロケーションに bound されている場合には，unbound な値もそのロケーションに bound される．両方の値が異なる具体ロケーションに bound されている場合には，制約を満たせないためランタイムエラーとなる．演算結果はオペランドとロケーション変数を共有するため，オペランドと同じロケーションに配置される．

ロケーション変数の間に追加された制約は，複数の値を同じロケーションへ配置するという関係を，後続の計算においても維持する．この設計では，ある値のロケーションが後から決定された場合にも，その値と同じロケーションに配置されるべきほかの値のロケーションも同時に決定される．

unbound な動的配置値を具体ロケーションを持つ変数に代入すると，その値のロケーション変数には，代入先の具体ロケーションと等しいという制約が追加される．これにより，その動的配置値は代入先の具体ロケーションに bound される．一方，すでに異なる具体ロケーションに bound された動的配置値を代入しようとした場合には，ロケーションの制約を満たせないためランタイムエラーとなる．

リスト 3 では，変数 `raw` は条件分岐で選択されたセンサに bound されているのに対し，リテラル `0.5@dyn` から生成された `bias` は初め unbound である．関数 `calibrate` 内の加算では，`raw` と `bias` が同じロケーションに配置される必要があるため，両者のロケーション変数が等しいという制約が追加される．その結果，`bias` は `raw` と同じセンサに bound される．加算結果 `calibrated` はオペランドとロケーション変数を共有するため，両オペランドと同じセンサに配置され，実際の加算もそのセンサで実行される．

ロケーション変数の間に追加された制約は，すべてのエンドポイントで整合する．各エンドポイントは同じ規則に従って制約を追加するため，ロケーション変数の具体ロケーションを決定すること自体には追加の通信を必要としない．ただし，これはすべてのエンドポイントが条件分岐において同じ分岐を選択することを前提としている．一方，加算や通信など，payload を操作する実際の演算は，制約追加後のロケーション

変数に基づいて選択されたエンドポイントだけが実行する．その他のエンドポイントは，制約を同様に追加するが，payload に対する演算は実行しない．この整合性により，各エンドポイントは，自身が演算を実行すべきかをロケーション変数とその制約に基づいて局所的に判断できる．次節では，この性質を利用して，動的配置値を含む式を各エンドポイントへ射影する方法を説明する．

3.3 ロケーションディスパッチ

全ての値のロケーションが静的に決まる式では，そのロケーションに基づいて，式を実行するエンドポイントをコンパイル時に決定できる．一方，動的配置値のロケーションは実行時まで決定されないため，動的配置値を含む式を特定のエンドポイントだけに射影することはできない．そこで，動的配置値を含む式を動的演算へ変換し，その動的演算を全エンドポイントへ射影する．例えば，リスト 3 の関数 `calibrate` 内の加算は，特定のセンサだけに射影されるのではなく，実行時に実行主体を決定する動的な加算演算として全エンドポイントへ射影される．

各エンドポイントは，実行時にロケーション変数を参照し，自身が動的演算を実行するかを局所的に決定する．本稿では，ロケーション変数に基づいて動的演算の実行主体を選択する処理を**ロケーションディスパッチ**と呼ぶ．

動的な加算演算では，オペランドが同じロケーションに配置されるように，それらのロケーション変数が等しいという制約を追加した後，ロケーションディスパッチを行う．ロケーション変数が具体ロケーション p に決定された場合には，ロケーション p に基づいて選択されたエンドポイントだけが加算を実行する．ロケーション変数が unbound のままである場合には，すべてのエンドポイントがオペランドの payload を持つため，各エンドポイントが同じ加算を実行する．例えば，リスト 3 では，`raw` と `bias` のロケーション変数が等しいという制約が追加される．`raw` は選択されたセンサに bound されているため，この制約によって `bias` も同じセンサに bound され，そのセンサだけが加算を実行する．加算結果 `calibrated` も両オペランドと同じロケーションに配置される．

通信式についても，コピー元とコピー先のロケーション変数に基づいて，payload の送信を担当するエンドポイント，受信するエンドポイント，および通信の要否を実行時に決定する．コピー先のエンドポイントがコピー元の payload を保持していない場合には，その payload を保持するエンドポイントからコピー先へ送信する．一方，コピー先がコピー元の payload をすでに保持している場合には，通信を行わずに値を局所的にコピーする．いずれの場合も，通信式の実行はコピー元のロケーション変数を変更せず，コピー先に新しい値を生成する．例えば，リスト 3 の 25 行目の通信式では，午前中には `calibrated` が `sensor1` に bound されているため，`sensor1` から `server` へ payload を送信する．午後には `calibrated` が `sensor2` に bound されているため，`sensor2` から `server` へ payload を送信する．いずれの場合にも，コピー元の動的配置値は変更されず，`server` に配置された新しい値が生成され，変数 `result` に束縛される．

ロケーションディスパッチでは，各エンドポイントが局所的に自身の役割を決定するが，その決定結果がすべてのエンドポイントで一致するように設計する．これは，ロケーション変数の状態と，それらの間に追加された制約が，すべてのエンドポイントで整合しているためである．各エンドポイントは同じ情報と規則に基づいてロケーションディスパッチを行うため，演算の実行主体や通信の送信元について同じ判断に到達する．例えば，リスト 3 の通信では，午前中にはすべてのエンドポイントが `calibrated` のロケーションを `sensor1` として認識する．したがって，`sensor1` は送信を実行し，`server` は `sensor1` からの受信を実行し，その他のエンドポイントは送信を実行しない．午後には，すべてのエンドポイントが `sensor2` を送信元として選択する．このように，送信側と受信側の判断に不一致が生じないため，ロケーションディスパッチの不一致に起因するデッドロックを生じさせないことを目指す．

$$
\begin{array}{lll}
L ::= p \mid \mathsf{dyn} & & \text{ロケーション} \\
T ::= b@L \mid T \to T & & \text{型} \\
v ::= c@p \mid \lambda x : T.\, C & & \\
C ::= x \mid v \mid C\, C & & \\
\quad \mid \ op(C, C) & & \\
\quad \mid \ \mathbf{let}\ x : b@L \leftarrow C\ \mathbf{in}\ C & &
\end{array}
$$

図 1　λ_{cbs} の構文

$$
\begin{array}{l}
v ::= c@p \mid \langle c, \alpha \rangle \mid \lambda x : T.\, A \\
A ::= x \mid v \mid A\, A \mid op(A, A) \\
\quad \mid \ \mathsf{dynop}(A, A) \\
\quad \mid \ \mathsf{inj}\langle p \rangle(A) \mid \mathsf{cast}\langle p \rangle(A) \\
\quad \mid \ \mathbf{let}\ x : b@L = A\ \mathbf{in}\ A \\
\quad \mid \ \mathsf{copy}\langle p, q \rangle(A) \\
\quad \mid \ \mathsf{dyncopy}\langle L \rangle(A)
\end{array}
$$

図 2　$\lambda_{\langle \mathrm{cbs} \rangle}$ の構文

4　λ_{cbs} と $\lambda_{\langle \mathrm{cbs} \rangle}$ による形式化

本章では，第 3 章で説明した漸進的ロケーティングの中核となる型付けと実行時機構を明確にするため，ソース言語 λ_{cbs} と，その実行時言語 $\lambda_{\langle \mathrm{cbs} \rangle}$ を定義する．λ_{cbs} は，ChoreoBS の言語機能のうち，具体ロケーションおよび動的ロケーションを持つ値，関数，プリミティブ演算，および通信を抽象化した言語である．$\lambda_{\langle \mathrm{cbs} \rangle}$ は，λ_{cbs} の式の変換先となる実行時言語であり，ロケーション変換やロケーションの検査，動的演算，および値のコピーを明示的に表す．

なお，本章で示すのは，形式化に向けた予備的な体系である．形式化の対象は，動的配置値の *unbound* 状態と *bound* 状態，制約に基づく実行時ロケーションの決定，および通信式による値のコピーであり，EPP，ロケーションディスパッチ，およびエンドポイント間の通信動作は考慮しない．また，すべてのエンドポイントに値を配置する *every* ロケーションも考慮しないこととし，具体ロケーションは単一のエンドポイントを指すものとする．第 3 章でランタイムエラーと呼んだロケーション制約の違反は，本章ではロケーションエラーとして形式化する．

4.1　λ_{cbs} の構文

λ_{cbs} の構文を図 1 に示す．λ_{cbs} は，各エンドポイントの動作を大域的に記述するコレオグラフィである．λ_{cbs} は漸進的ロケーティングを取り入れており，値のロケーション L として，具体ロケーション p と動的ロケーション dyn を扱う．L に配置された基底型 b の値の型を $b@L$ で表し，L に配置された定数 c を $c@L$ で表す．関数にはロケーションを付与せず，すべての関数を大域的に使用できるものとする．式 $op(C_1, C_2)$ は，二つの式 C_1 と C_2 の評価結果にプリミティブ演算 op を適用することを表す．

通信式 $\mathbf{let}\ x : b@L \leftarrow C_1\ \mathbf{in}\ C_2$ は，C_1 の評価結果の値からロケーション L に配置される新しい値を生成し，その値を x に束縛して C_2 を評価することを表す．通信式は値のコピーを抽象化したものであり，コピー元とコピー先のロケーションに応じて，エンドポイント間通信または同一エンドポイント内のコピーとして実行される．なお，通常の束縛 $\mathbf{let}\ x = C_1\ \mathbf{in}\ C_2$ は関数適用の糖衣構文とみなせるため，λ_{cbs} の構文には含めない．

4.2　$\lambda_{\langle \mathrm{cbs} \rangle}$ の構文と型付け

λ_{cbs} の実行時の振る舞いを議論するための基盤として，実行時言語 $\lambda_{\langle \mathrm{cbs} \rangle}$ を導入し，その構文と型付け規則を図 2 と図 3 に示す．$\lambda_{\langle \mathrm{cbs} \rangle}$ の式は，λ_{cbs} の式を型付けに基づいて変換することで得られる．$\lambda_{\langle \mathrm{cbs} \rangle}$ では，静的配置値と動的配置値の間の変換や，動的配置値に対する検査および操作を明示的に表す．これにより，動的配置値に対する処理を静的配置値に対する処理から分離し，それぞれの実行時の振る舞いを個別に定義できる．

動的配置値 $\langle c, \alpha \rangle$ は，値の本体を表す payload c とロケーション変数 α の組である．具体ロケーション p を持つ値を静的配置値と呼び，λ_{cbs} と同様に $c@p$ によって表現する．定数 c の基底型は，λ_{cbs} と共通のシグネチャ Δ によって $\Delta(c) = b$ と与える．

静的配置値と動的配置値の間の変換は，$\mathsf{inj}\langle p \rangle$ と $\mathsf{cast}\langle p \rangle$ によって表す．$\mathsf{inj}\langle p \rangle(A)$ は，ロケーション p を持つ静的配置値を動的配置値へ変換する．$\mathsf{cast}\langle p \rangle(A)$ は，動的配置値のロケーションが p であることを実行時に確定させ，その値をロケーション p を持つ静的配置値へ変換する．

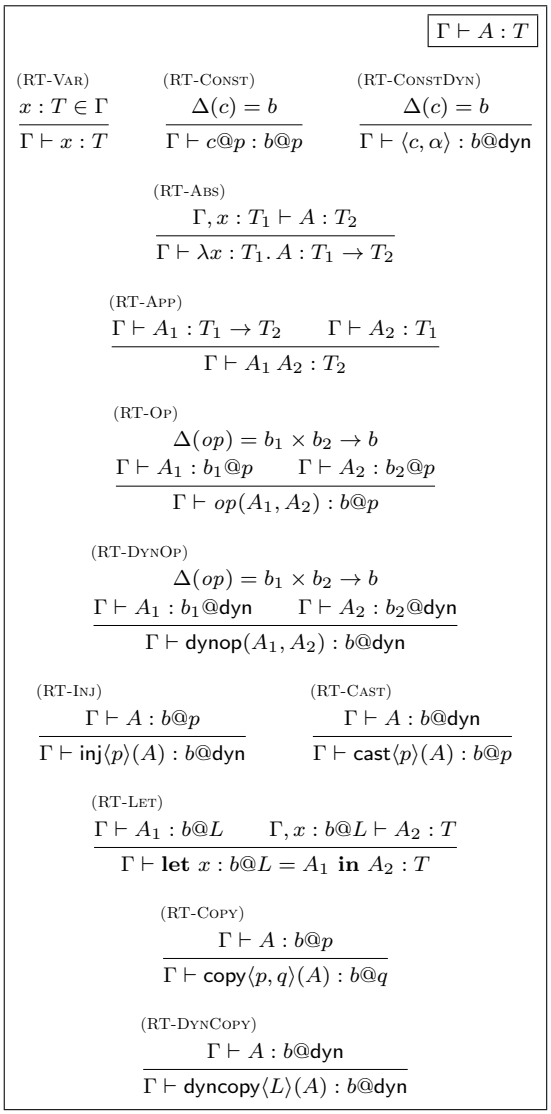

図 3　$\lambda_{\langle \mathrm{cbs} \rangle}$ の型付け規則

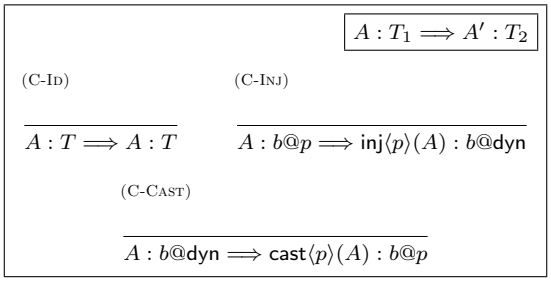

図 4　ロケーション変換規則

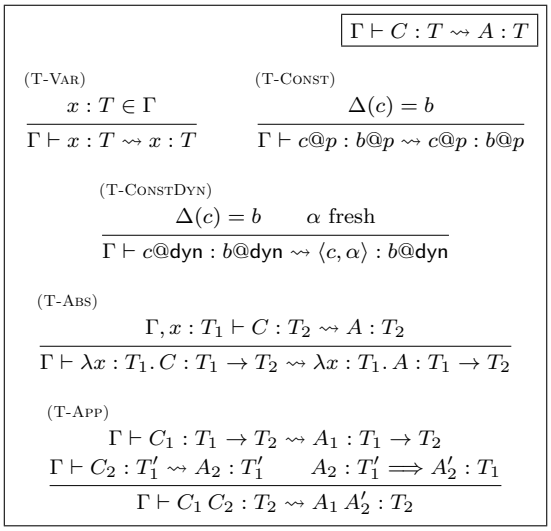

図 5　基本式および関数の型付けと変換規則

$\mathsf{dynop}(A_1, A_2)$ は，動的配置値に対するプリミティブ演算を表す．動的，静的にかかわらず，すべての演算でオペランドは同一のロケーションになければならないので，dynop は二つのオペランドのロケーションが等しいことを実行時に確定させる．これに対して，$op(A_1, A_2)$ は，ロケーションが静的に決まっている値に対するプリミティブ演算を表す．op の型付けは二つのオペランドが同一の具体ロケーションを持つことを要求するため，ロケーションが異なる場合には型付けできない．プリミティブ演算 op の引数と結果の基底型は，λ_{cbs} と共通のシグネチャ Δ によって $\Delta(op) = b_1 \times b_2 \to b$ と与える．

$\mathsf{dyncopy}\langle L \rangle(A)$ は，動的配置値の payload をロケーション L へコピーし，新しい動的配置値を生成する．これに対して，$\mathsf{copy}\langle p, q \rangle(A)$ は，ロケーション p を持つ静的配置値をロケーション q へコピーする．また，λ_{cbs} の通信式を dyncopy または copy を用いた式へ変換するため，$\lambda_{\langle \mathrm{cbs} \rangle}$ には $\mathbf{let}\ x : b@L = A_1\ \mathbf{in}\ A_2$ を設ける．

4.3　λ_{cbs} の型付けと $\lambda_{\langle \mathrm{cbs} \rangle}$ への変換

λ_{cbs} の型付け規則を図 5 と図 6 に示す．簡略化のため、well-typed な λ_{cbs} の式から $\lambda_{\langle \mathrm{cbs} \rangle}$ への変換規則も図 5 と図 6 に埋め込む．ソース言語の型付けと実行時言語への変換を，判断 $\Gamma \vdash C : T \rightsquigarrow A : T$ によって表す．これは，型環境 Γ の下でソース式 C が

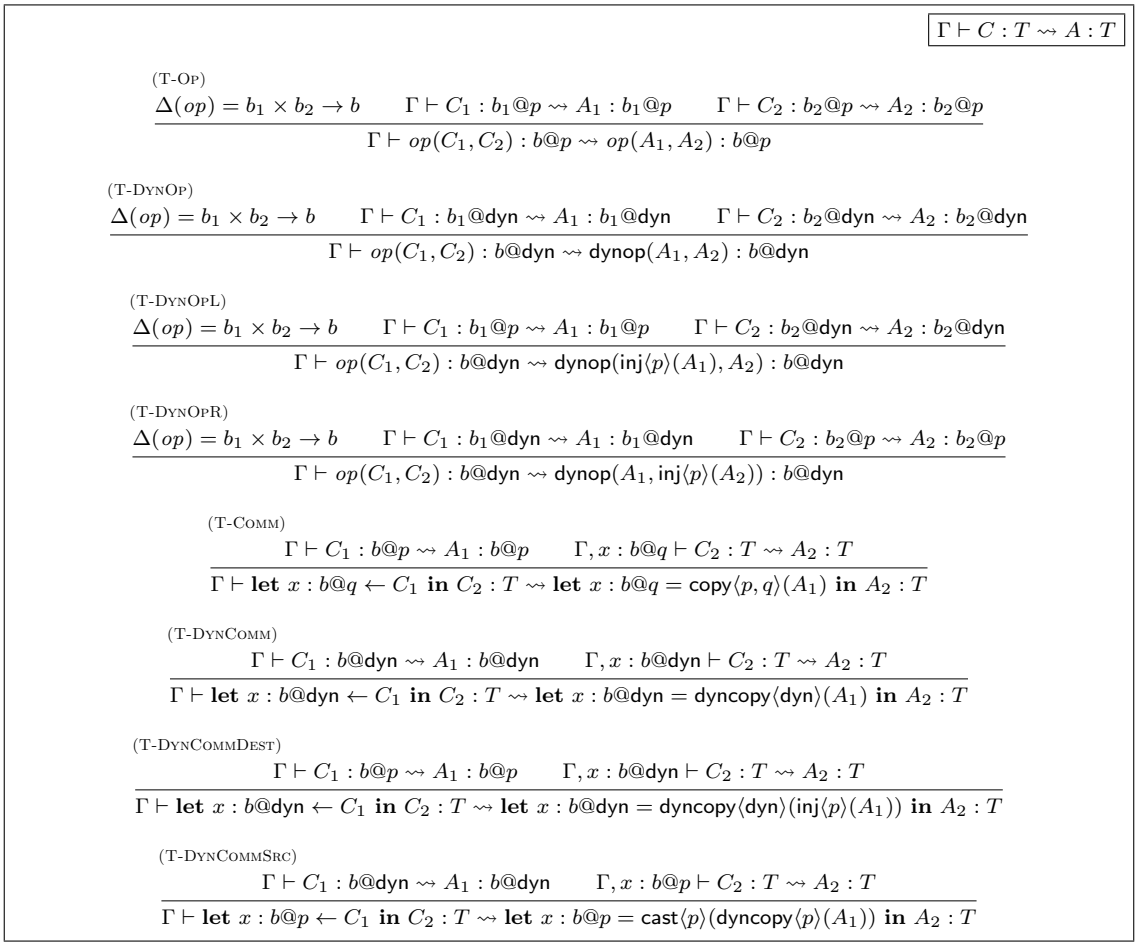

図 6　プリミティブ演算および通信の型付けと変換規則

型 T を持ち，実行時式 A へ変換されることを意味する．変換後の式 A に付与したこの型 T が，図 3 の $\lambda_{\langle\mathrm{cbs}\rangle}$ の型付け規則によっても A に与えられることは，定理 4.1 で示す．

図 5 に基本式および関数の型付けと変換規則を示す．規則 T-ConstDyn が示すように，$c@\mathsf{dyn}$ は $\langle c, \alpha\rangle$ に変換される．α fresh は，生成される実行時式中のほかのロケーション変数と α が重複しないことを意味する．

関数適用の変換規則は T-App である．関数適用では，実引数の型が仮引数の型に合うように，図 4 のロケーション変換規則に従って変換を挿入する．具体ロケーションを持つ値を動的ロケーションの仮引数に渡す場合は inj を挿入し，逆の場合は cast を挿入する．型が一致する場合には，実引数をそのまま用いる．ただし，ソース式が well-typed であることを前提とするため，仮引数と実引数がともに具体ロケーションを持つ場合にはそれらは一致しており，異なる具体ロケーション間の変換（$b@p$ から $b@q$，$p \neq q$）は生じない．

図 6 にプリミティブ演算および通信の型付けと変換規則を示す．プリミティブ演算では，二つのオペランドがともに具体ロケーションを持つ場合，ソース式が well-typed であることからそれらは同一のロケーション p であり，T-Op により静的な演算 op へ変換する．両方または一方が動的ロケーションを持つ場合には，T-DynOp，T-DynOpL，T-DynOpR により dynop へ変換し，必要に応じて inj を挿入する．通信式の変換には，コピー元とコピー先がともに具体ロケーションの場合は T-Comm，ともに動的ロケー

ションの場合は T-DynComm，具体ロケーションから動的ロケーションへの場合は T-DynCommDest，動的ロケーションから具体ロケーションへの場合は T-DynCommSrc を用いる．

4.4 制約ストア

$\lambda_{\langle \mathrm{cbs} \rangle}$ の操作的意味論では，動的配置値のロケーション変数に対して実行中に得られた等式制約を管理するため，制約ストア Σ を用いる．制約の対象となる具体ロケーション p とロケーション変数 α を総称してロケーション項 ℓ と呼び，次のように定義する．

$$\ell ::= p \mid \alpha$$

二つのロケーション項が同じロケーションを表すべきであるという等式制約を，

$$\ell_1 = \ell_2$$

の形で表す．制約ストア Σ は，このような等式制約を保持するものであり，次のように定義する．

$$\Sigma ::= \emptyset \mid \Sigma, \ell_1 = \ell_2$$

ここで，$\emptyset$ は空の制約ストアを表し，$\Sigma, \ell_1 = \ell_2$ は，Σ に等式制約 $\ell_1 = \ell_2$ を追加したストアを表す．以降では，制約ストアを等式制約の集合として扱い，制約の追加を $\Sigma \cup \{\ell_1 = \ell_2\}$ とも書く．

判断

$$\Sigma \vdash \ell_1 = \ell_2$$

は，Σ に格納された等式制約に反射律，対称律，および推移律を適用することで，ℓ_1 と ℓ_2 が同じロケーションを表すと導出できることを意味する．例えば，$\alpha = \beta$ と $\beta = p$ が Σ に含まれるならば，$\Sigma \vdash \alpha = p$ が成立する．

制約ストアから異なる二つの具体ロケーションが等しいと導出できる状態を，ロケーション衝突と呼ぶ．制約ストアにロケーション衝突がないことを衝突自由性と呼び，次のように定義する．

$$\mathsf{conflictFree}(\Sigma) \iff \forall p, q.\ \Sigma \vdash p = q \Longrightarrow p = q$$

すなわち，Σ から二つの具体ロケーションが等しいと導出できるならば，それらは実際に同一の具体ロケーションでなければならない．

二つのロケーション項を同一視する制約を制約ストアへ反映する操作を，unify として次のように定義する．

$$\begin{array}{rcl} E & ::= & [\,] \mid E\ A \mid v\ E \\ & \mid & op(E, A) \mid op(v, E) \\ & \mid & \mathsf{dynop}(E, A) \mid \mathsf{dynop}(v, E) \\ & \mid & \mathsf{inj}\langle p \rangle(E) \mid \mathsf{cast}\langle p \rangle(E) \\ & \mid & \mathbf{let}\ x : b@L = E\ \mathbf{in}\ A \\ & \mid & \mathsf{copy}\langle p, q \rangle(E) \\ & \mid & \mathsf{dyncopy}\langle L \rangle(E) \end{array}$$

図 7　評価文脈

$$\mathsf{unify}(\Sigma, \ell_1, \ell_2) = \begin{cases} \Sigma & \text{if } \Sigma \vdash \ell_1 = \ell_2, \\ \Sigma \cup \{\ell_1 = \ell_2\} & \text{if } \mathsf{conflictFree}(\Sigma \cup \{\ell_1 = \ell_2\}), \\ \mathsf{fail} & \text{otherwise.} \end{cases}$$

以降では，この操作を単一化と呼ぶ．すでに ℓ_1 と ℓ_2 が等しいと導出できる場合，制約ストアは変更しない．まだ導出できない場合は，ロケーション衝突が生じない限り，新しい等式制約を追加する．制約の追加によって異なる具体ロケーションが同一視される場合，単一化は失敗する．

動的配置値の *bound* 状態と *unbound* 状態は，値の構文だけでは決まらず，現在の制約ストアにも依存して定義する．

$$\mathsf{bound}_\Sigma(\alpha, p) \iff \Sigma \vdash \alpha = p$$

$$\mathsf{unbound}_\Sigma(\alpha) \iff \nexists p.\ \Sigma \vdash \alpha = p$$

したがって，動的配置値 $\langle c, \alpha \rangle$ について，$\Sigma \vdash \alpha = p$ が成立するならば，この値は p へ束縛されており，*bound* である．一方，α と等しい具体ロケーションを導出できない場合，この値は *unbound* である．unbound であることは payload が未計算であることを意味せず，payload c の論理的な配置先だけが未決定であることを意味する．

補題 4.1 (単一化による衝突自由性の保存)．$\mathsf{conflictFree}(\Sigma)$ かつ $\mathsf{unify}(\Sigma, \ell_1, \ell_2) = \Sigma'$ ならば，$\mathsf{conflictFree}(\Sigma')$ である．

補題 4.2 (fresh 変数の束縛による衝突自由性の保存)．$\mathsf{conflictFree}(\Sigma)$ かつ α が Σ に現れないならば，$\mathsf{conflictFree}(\Sigma \cup \{\alpha = p\})$ である．

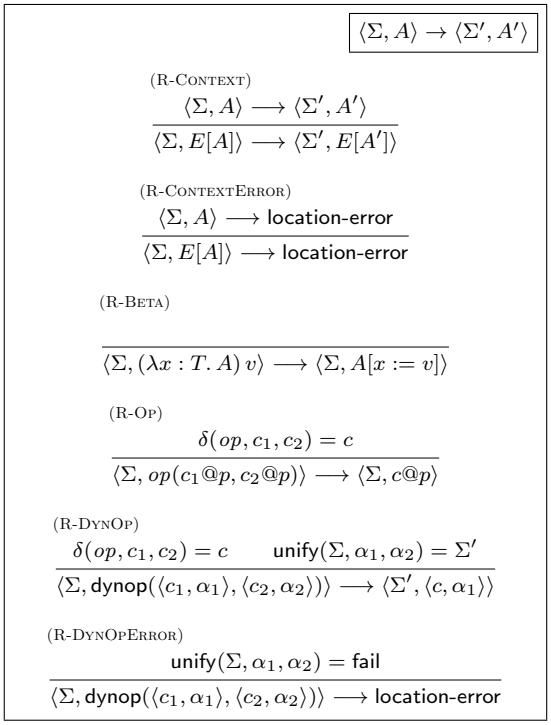

図 8　文脈および演算の簡約規則

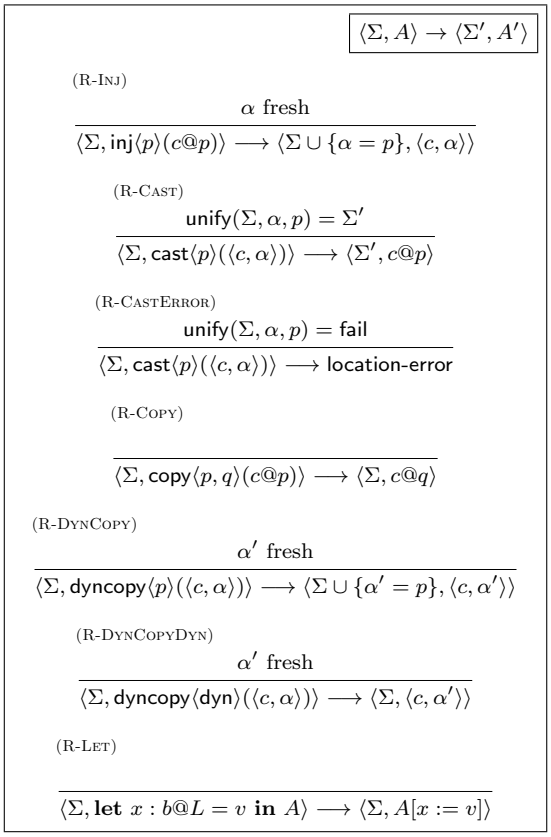

図 9　変換・コピーおよび束縛の簡約規則

4.5　操作的意味論

$\lambda_{\langle \mathrm{cbs}\rangle}$ は左から右への評価順序を持つ call-by-value で評価する．評価関係を $\langle \Sigma, A\rangle \longrightarrow \langle \Sigma', A'\rangle$，ロケーション衝突による実行の中断を $\langle \Sigma, A\rangle \longrightarrow \mathsf{location\text{-}error}$ と書く．評価文脈を図 7 に，簡約規則を図 8 および図 9 に示す．

プリミティブ演算における payload の計算は，メタレベルの関数 δ によって与える．δ は型に適合する入力について全域であり，型を保存すると仮定する．すなわち，$\Delta(op) = b_1 \times b_2 \to b$，$\Delta(c_1) = b_1$，および $\Delta(c_2) = b_2$ ならば，ある定数 c が存在して，

$$\delta(op, c_1, c_2) = c \qquad \text{かつ} \qquad \Delta(c) = b$$

が成立するものとする．

dynop では，二つのオペランドのロケーション変数を単一化した後に payload を計算する．単一化の後には $\Sigma' \vdash \alpha_1 = \alpha_2$ が成立するため，結果のロケーション変数には代表元として α_1 を用いる．これにより，演算結果のロケーションはオペランドのロケーションと一致する．単一化に失敗する場合にはロケーションエラーとなる．

inj は fresh なロケーション変数 α を生成し，制約 $\alpha = p$ を制約ストアに追加することで，静的配置値の具体ロケーションを記録する．fresh な変数への制約の追加は制約ストアの衝突自由性を保つため（補題 4.2），inj は失敗しない．cast は動的配置値のロケーション変数を具体ロケーション p と単一化する．したがって，cast はすでに p へ束縛された値を検査するだけでなく，unbound な値を p へ新たに束縛する．

dyncopy は，fresh なロケーション変数 α' を持つ新しい動的配置値を生成する．コピー先が具体ロケーション p の場合には制約 $\alpha' = p$ を追加し，動的ロケーションの場合には制約を追加せず α' は unbound のままとなる．いずれのコピー規則もコピー元のロケーション変数 α を変更せず，α に対する制約も追加しない．copy はコピー元の payload をコピー先 q に配置した新しい静的配置値を生成する．

4.6 型安全性

本節では，前節までに定義した型付け，変換，および操作的意味論について型安全性を示す．まず，λ_{cbs} から $\lambda_{\langle\mathrm{cbs}\rangle}$ への変換が型を保存することを示す．次に，$\lambda_{\langle\mathrm{cbs}\rangle}$ の評価について，評価の前後で式の型と制約ストアの衝突自由性が保存されることを示す．最後に，well-typed な閉じた式は，値であるか，評価を継続できるか，ロケーションエラーとなるかのいずれかであることを示す．

λ_{cbs} のソース式に動的ロケーションが含まれる場合，変換後の $\lambda_{\langle\mathrm{cbs}\rangle}$ の式では，具体ロケーションに関する検査の一部が実行時に行われる．そのため，well-typed な実行時式であってもロケーションエラーとなり得る．したがって，本節ではロケーションエラーを許容した上で，ロケーション衝突が明示的にロケーションエラーとして検出され，それ以外の理由では $\lambda_{\langle\mathrm{cbs}\rangle}$ の評価が行き詰まらないことを型安全性として示す．

補題 4.3 (ロケーション変換の型付けの健全性)．$\Gamma \vdash A : T_1$ かつ $A : T_1 \Longrightarrow A' : T_2$ ならば，$\Gamma \vdash A' : T_2$ である．

定理 4.1 (変換による型の保存)．$\Gamma \vdash C : T \rightsquigarrow A : T$ ならば，図 3 の $\lambda_{\langle\mathrm{cbs}\rangle}$ の型付け規則の下で $\Gamma \vdash A : T$ が導出できる．

証明は，$\Gamma \vdash C : T \rightsquigarrow A : T$ の導出に関する帰納法による．関数適用の場合には補題 4.3 を用いる．

定理 4.2 (簡約による型と衝突自由性の保存)．$\Gamma \vdash A : T$，$\mathsf{conflictFree}(\Sigma)$，および $\langle \Sigma, A\rangle \longrightarrow \langle \Sigma', A'\rangle$ が成立するならば，$\Gamma \vdash A' : T$ かつ $\mathsf{conflictFree}(\Sigma')$ である．

証明は評価の導出に関する帰納法による．dynop および cast の場合には，補題 4.1 によって，更新後の制約ストアにもロケーション衝突がないことが保証される．inj および具体ロケーションへの dyncopy の場合には，fresh な変数への制約の追加であるため，補題 4.2 によって，同様のことが保証される．

前述のとおり，well-typed な式であってもロケーションエラーとなり得る．この点を考慮した進行性を次のように定める．

定理 4.3 (ロケーションエラーを除く進行性)．$\emptyset \vdash A : T$ かつ $\mathsf{conflictFree}(\Sigma)$ であるとき，次のいずれかが成立する．

1. A は値である．
2. ある Σ' および A' について，$\langle \Sigma, A\rangle \longrightarrow \langle \Sigma', A'\rangle$ である．
3. $\langle \Sigma, A\rangle \longrightarrow \mathsf{location\text{-}error}$ である．

この定理は，well-typed な閉じた式が，ロケーション衝突以外の理由で行き詰まらないことを表す．

5 関連研究

Siek と Taha は，型の未知な部分を表す動的型を導入し，静的型付けと動的型付けの間を段階的に移行可能にする漸進的型付けを提案した [10]．本研究の漸進的ロケーティングは，この考え方をベース型ではなくロケーション情報へ適用し，具体ロケーションに関する検査の一部を静的検査から実行時検査へ移す．また，Siek らは，型注釈の精度だけが異なるプログラム間の静的および動的な振る舞いを関係付ける gradual guarantee を，漸進的型システムの評価基準として定式化した [11]．本研究では，部分的なロケーションの抽象化によって実行結果を変えず，静的なロケーション検査を実行時検査へ移行できることを設計目標とするが，その証明は今後の課題とする．

Samuelson らの λ_{QC} は，ロケーション名およびロケーション集合を first-class に扱う型付きコレオグラフィ言語であり，ローカル計算によって実行時に選択されたロケーション名を参加者間で共有し，後続の計算主体や通信相手として利用できる [9]．これに対し，本研究の漸進的ロケーティングは，動的配置値が配置先に関する情報を実行時に保持し，動的演算はその情報に基づいて実行主体を決定する．そのため，プログラマがロケーション名を明示的に計算および共有し，後続の計算において指定することなく，異なるロケーションで生成された値を同一の動的配置変数として扱うことができる．さらに，配置先が未決定の値を，後続の演算が要求する共配置関係に基づいて遅延して具体化できる点，および静的ロケーションと動的ロケーションを同一プログラム内で混在させ，配置情報を段階的に具体化できる点で，λ_{QC} とは異なる．

通常の値型以外の静的性質に漸進的型付けを適用

した研究として，gradual security types と gradual session types がある．Fennell と Thiemann は，Java 風のオブジェクト指向言語において，具体的なセキュリティレベルを持つ部分を静的に検査する一方，動的セキュリティ型を持つ部分では実行時セキュリティラベルを伝播および検査する LJGS を提案している [3]．また，Igarashi らは，セッション型に漸進的型付けを適用した Gradual GV を提案し，通信プロトコルを動的に検査するとともに，動的コード内でもチャネルの線形性を維持するため，線形値の使用状態を管理する実行時機構を導入している [7]．これらの研究は，静的な注釈を単に未知の注釈へ置き換えるだけでなく，対象領域に固有の保証を動的な部分でも維持するための実行時情報と検査機構が必要になることを示している．本研究も同様に，漸進性をロケーション情報へ適用するため，動的配置値の具体的な配置先が未決定か決定済みかを表す *unbound* および *bound* 状態を導入する．さらに，実行時に決定された配置先に基づいて演算主体および通信主体を選択するロケーションディスパッチを導入する点で，セキュリティレベルや通信プロトコルを漸進化する既存研究とは異なる．

6 まとめ

本研究では，コレオグラフィックプログラミングのための漸進的ロケーティングを提案した．静的ロケーションと動的ロケーションを組み合わせることで，異なるロケーションで生成された値を分岐後に統一的に扱い，共通する処理の重複を削減できる．また，動的配置値に *unbound* 状態と *bound* 状態を導入し，共配置制約に基づいて値の配置先を実行時に決定する仕組みを示した．さらに，すべてのエンドポイントで整合したロケーション情報に基づいて，演算および通信の実行主体を選択するロケーションディスパッチを示した．

謝辞 本研究は、JST 次世代研究者挑戦的研究プログラム JPMJSP2108 の支援を受けたものです。また，本研究は JSPS 科研費 24H00688 の助成を受けたものです。

参考文献

[1] Bates, M. and Near, J. P.: We Know I Know You Know: Choreographic Programming with Multicast and Multiply Located Values, 2024. Presented at the 1st International Workshop on Choreographic Programming (CP 2024).

[2] Carbone, M. and Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming, *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, New York, NY, USA, Association for Computing Machinery, 2013, pp. 263–274.

[3] Fennell, L. and Thiemann, P.: LJGS: Gradual Security Types for Object-Oriented Languages, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, Krishnamurthi, S. and Lerner, B. S.(eds.), Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56, Dagstuhl, Germany, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016, pp. 9:1–9:26.

[4] Giallorenzo, S., Montesi, F., and Peressotti, M.: Choral: Object-oriented Choreographic Programming, *ACM Trans. Program. Lang. Syst.*, Vol. 46, No. 1(2024).

[5] Graversen, E., Hirsch, A. K., and Montesi, F.: Alice or Bob?: Process polymorphism in choreographies, *J. Funct. Program.*, Vol. 34(2024).

[6] Hirsch, A. K. and Garg, D.: Pirouette: higher-order typed functional choreographies, *Proc. ACM Program. Lang.*, Vol. 6, No. POPL(2022).

[7] Igarashi, A., Thiemann, P., Vasconcelos, V. T., and Wadler, P.: Gradual session types, *Proc. ACM Program. Lang.*, Vol. 1, No. ICFP(2017).

[8] Montesi, F.: *Choreographic Programming*, PhD Thesis, IT University of Copenhagen, August 2013.

[9] Samuelson, A., Hirsch, A. K., and Cecchetti, E.: Choreographic Quick Changes: First-Class Location (Set) Polymorphism, *Proc. ACM Program. Lang.*, Vol. 9, No. OOPSLA2(2025).

[10] Siek, J. G. and Taha, W.: Gradual Typing for Functional Languages, *Scheme and Functional Programming Workshop*, 2006, pp. 81–92.

[11] Siek, J. G., Vitousek, M. M., Cimini, M., and Boyland, J. T.: Refined Criteria for Gradual Typing, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, Ball, T., Bodík, R., Krishnamurthi, S., Lerner, B. S., and Morriset, G.(eds.), Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32, Dagstuhl, Germany, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015, pp. 274–293.