差分実行の形式化

指導教員: 増原英彦教授

2022 年度修了予定

古殿直也

21M30353

概要

Kanon はライブプログラミング環境の一つで、プログラムを実行した際に作られるデータ構造を可視化する。プログラムを編集するごとに、即座にデータ構造の可視化を更新することで、プログラムの編集がどのような影響を与えるかをプログラマが理解しやすくする。

Kanon ではデータ構造を描画するために、プログラムの編集が起きる度にプログラムを実行し、実行時情報を収集する。プログラムの実行に時間がかかると、編集と描画結果のの間に一貫性がなくなり、プログラマの理解が妨げられる。

そこで本研究ではプログラムの実行にかかる時間を減らすための手法として、プログラムの差分実行方式を提案する。差分実行方式でプログラムを実行する際には、(1) プログラムの実行履歴を収集し、(2) 前回の実行履歴を用いて、前回と実行結果が同じになる箇所を判定し、(3) 実行履歴を用いて計算状況を書き換えることで実行の一部をスキップする。

Kanon のように類似のプログラムを連続して実行する場合、差分実行方式はプログラム実行の大部分を省略できると見込まれ、Kanon の即応性を向上することを期待できる。

その一方で通常のインタプリタに比較して、差分実行方式を採用する場合履歴の収集やプログラム実行の差分の判定にオーバーヘッドがかかる。それらを安全に、かつ効率的に処理するために安全性の検証や、差分実行方式の最適化が必要である。そのために本研究では、差分実行方式をプログラミング言語の操作的意味論として形式化し、その安全性を定式化する。形式化を通じて差分実行方式の安全性を保証するために必要な実行履歴の性質を見出すことで、実装の正しさの保証や最適化手法の検討に貢献できると考える。

第 1 章

1. 導入

ライブプログラミング環境はプログラマが入力したソースコードに対して、実行結果を表示したり、システムの変更を反映したりすることでプログラミングやプログラムの理解を支援する。プログラミングによる音楽のライブパフォーマンスを支援する Sonic Pi[1] や、データ構造プログラミングを支援する Kanon[5] が例として挙げられる。

ライブプログラミング環境はソースコードの編集に対してすぐさま応答することが重要である。それは編集の影響がすぐさまフィードバックされることで、プログラマはプログラムの変更と振る舞いの変化の間の因果関係を理解しやすくなる [7] からだ。

ライブプログラミング環境において変更に対する応答速度を向上するための手法が研究されている。Rein et al. [6] はプログラマが変更を与えてから、その影響がユーザに観測できるようになるまでの時間をシステム応答時間 (system response time) とよび、システム応答時間を 2 つの段階に分けて分析した。システム応答時間の前半を受容段階 (adaption phase) と呼び、後半を出現段階 (emergence phase) と呼ぶ。受容段階はプログラマがソースコードの編集を終えてから、その実行のための形式に変換し、実行の準備が整うまでの期間を指す。出現段階は実行を開始してから、それによるシステムの振る舞いの変化が現れるまでの期間を指す。システム応答時間を短く保つためには、受容段階と出現段階の両方を短く保つ必要がある。

本研究ではライブプログラミング環境 Kanon の出現段階にかかる時間を短縮することを目的として、プログラムの差分実行方式を提案する。差分実行方式では、プログラムを実行する際に実行情報を保存する。編集後のプログラムを実行する際に、差分だけを実行することで実行時間の短縮を狙う。

安全で効率のよい差分実行方式を実現する手法は非自明であり、安全性の証明や最適化手法を検討するためのコンパクトな定義があると望ましい。そこで本研究では、差分実行方式を小さなプログラミング言語の操作的意味論として定義する。差分実行方式を表す意味論と、通常の実行方式を表す意味論を比較することで差分実行方式の安全性を表現し、証明する。

以降の章は以下のように構成される。

- 2 章では本論文の背景を説明する。
- 3 章では Kanon の動作と差分実行の関係を述べた上で、差分実行方式を説明する。
- 4 章では差分実行方式を形式的な言語の意味論として定義し、安全性を議論する。
- 5 章では関連研究を本研究と比較する。
- 6 章では結論と今後の課題を述べる。

第 2 章

2. 背景

2.1. Kanon の振る舞い

Kanon[5] はプログラマが入力したプログラムで生成されるデータ構造を可視化するライブプログラミング環境である。図 1 は Kanon の操作画面である[1]。Kanon は以下のように動作する。

1. プログラマは画面左側のエディタにプログラムを入力する。
2. Kanon は入力されたプログラムに実行時情報を記録するためのコードを挿入する。
3. Kanon はプログラムを実行し、実行時情報を収集する。
4. Kanon は実行時情報をもとに、プログラマが指定したプログラム実行時点に存在するデータ構造を画面右側に描画する。

この一連の処理をプログラマがプログラムを編集するたびに行う。即座にプログラム編集の結果を描画することで、Kanon は複雑なデータ構造プログラミングを支援する。

図 2 は Kanon に入力として与えるプログラムと、その描画結果の列である。はじめに図 2a のプログラ

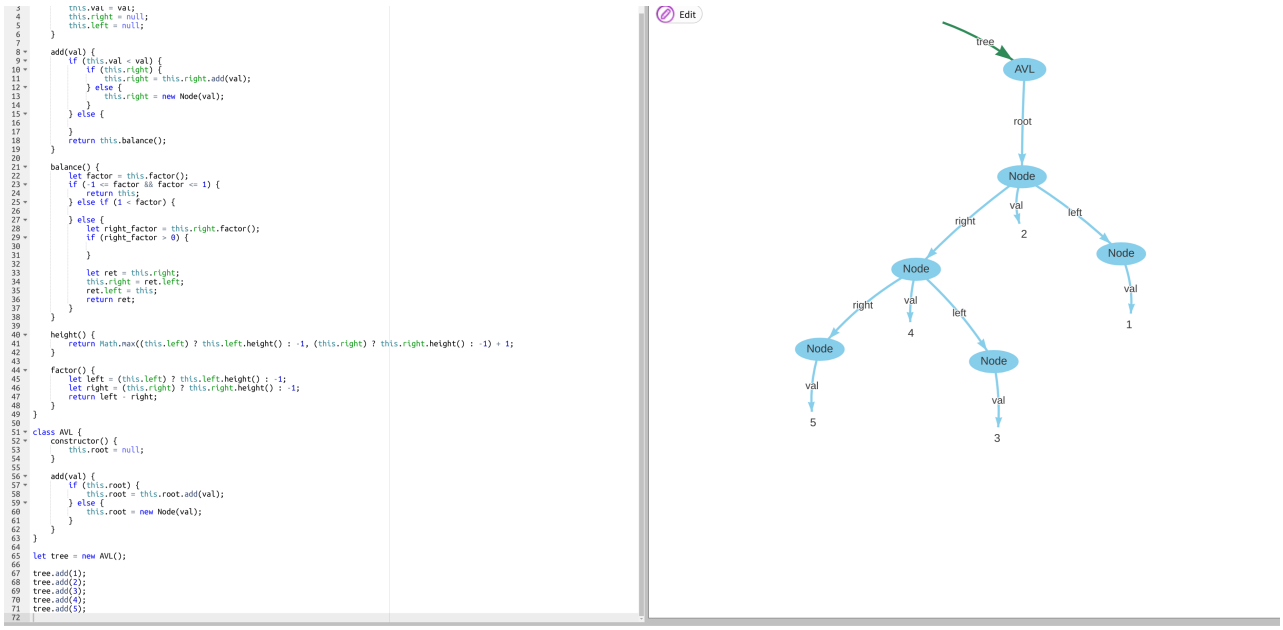


図 1. Kanon の操作画面

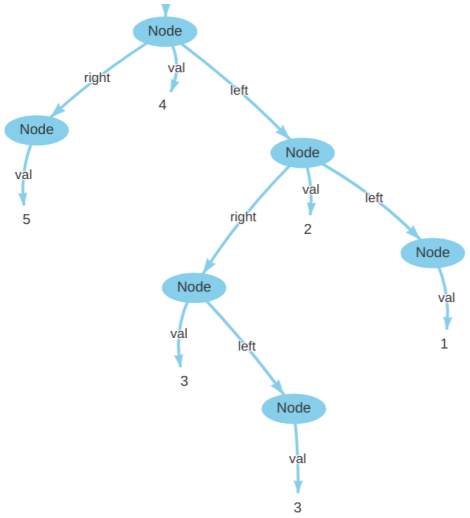
[1] https://prg-titech.github.io/Kanon/

```
let tree = new AVL();

tree.add(1);
tree.add(2);
tree.add(3);
tree.add(4);
tree.add(5);

let avg = tree.sum() / tree.size();
tree.add(avg);
```

(a)



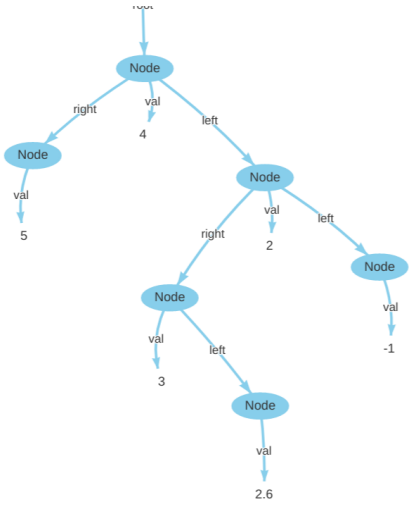
(b)

```
let tree = new AVL();

tree.add(-1);
tree.add(2);
tree.add(3);
tree.add(4);
tree.add(5);

let avg = tree.sum() / tree.size();
tree.add(avg);
```

(c)



(d)

図 2. Kanon の動作例

ムを入力し、最終行を描画のために指定したとき、Kanon はその時点で存在するオブジェクトを図 2b のように描画する。

その後プログラマがコードを編集し、図 2c のように書き換えた際は、Kanon は変更後のプログラムを**再実行**し、指定された時点のデータ構造を図 2d のように描画する。この例では `tree.add(1)` を `tree.add(-1)` に書き換えており、その結果として木構造の要素の値が一部変化している。Kanon を用いることで、プログラムの編集によりデータ構造のどの箇所が変化したかを即座に把握できる。

第 3 章

3. 差分実行方式

2 の例ではプログラムの編集によってデータ構造の要素の値だけが書き換えられており、オブジェクトの構造は変化していない。

そのため変数 avg の値を計算する箇所で、tree オブジェクトに対して tree.sum メソッドと tree.size メソッドを呼び出す。tree.sum メソッドと tree.size メソッドはともに、木構造を探索する。tree.sum はそれぞれのノードの値の総和をとり、tree.size は頂点数を計算する。

tree.sum の呼び出しの中では、初回の実行と二回目の実行の間で対応するオブジェクトのフィールドに異なる値が代入されており、結果として tree.sum() の値が変化する一方で、tree.size の呼び出しの中では、プログラム編集の影響を受けないデータしか計算に使わないため、初回の実行と二回目の実行の間で差異が存在しない。

このように編集されたプログラムを実行するなかで、編集前のプログラムの実行と異なる部分と同じ部分がある。このような着眼点で見た際の、プログラム実行の異なる部分を本研究では**プログラム実行の差分**と呼ぶ。2 の例では tree.sum メソッド呼び出しの箇所にはプログラム実行の差分があるが、tree.size メソッドの呼び出し箇所にはプログラム実行の差分がない。

初回の実行で必要な実行履歴を収集し、差分のない箇所は前回の実行履歴から結果を得ることで、二回目の実行の一部をスキップできる。このようにプログラムの実行の差分だけを実際に実行する実行方式をプログラムの**差分実行方式**と呼ぶ。差分実行方式は@takahashi:master に実装手法が提案されている。

3.1. 差分実行方式の意義

プログラム実行の差分が小さくなるような編集がプログラマから与えれた場合、差分実行方式では 2 回目以降のプログラム実行の大部分をスキップできるため、実行時間を短縮できることが見込まれる。このような実行時間の短縮が差分実行方式を採用する目的である。

Kanon をはじめとするライブプログラミング環境では、このような差分が小さくなるような編集が頻繁に入力されると期待できる。ライブプログラミング環境では、プログラムとそれに対するフィードバック（Kanon の場合はデータ構造の描画）が常に一貫性をもつことを目指しており、そのためにプログラムの編集が起きた際にすぐさまプログラムを実行する。そのため編集の量が小さくなり、プログラム実行の差分も小さくなると期待できる。

3.2. 差分実行の課題と形式化の意義

差分実行はライブプログラミング環境などの言語処理系として用いることで応答速度を高める期待がある一方で、一般的な言語処理系に比べて実装が複雑になる。差分実行方式を実現するためには、

- 実行時に実行履歴を収集すること、
- 前回実行時の実行履歴と、編集後のプログラムの実行時情報を元に差分があるか否かを判定すること、
- 差分がない場合に、前回実行時の実行履歴をもとに、実行をスキップし実行時情報を編集すること

が必要である。これらの仕組みを正しく効率的に動作するための実装は複雑にならざるを得ないだろう。

そこで本研究では差分実行方式にコンパクトな形式化を与えその正しさを証明する。コンパクトな定義によって、最適化の議論がしやくなり、正しさの証明があることで、保つべき不変条件が明らかになることを期待する。

第 4 章

4. 差分実行方式の形式化

この節では差分実行をする言語の定義を行う。はじめに言語の構文や操作的意味論を定義する際に用いるデータ構造を定義する。次に差分実行を含む、言語の操作的意味論を定義する。そして差分実行方式の安全性を定式化する。

4.1. 構文

図 3 で差分実行の形式化で考える言語の構文や、差分実行で用いるデータ構造を定義する。

4.2. 差分実行の評価規則

図 4 で差分実行の意味論を定義する。$(use\,history)$ 規則が適用できる際には、暗に他の規則を適用できないことを仮定している。

Expression	$e \in Expr$	|	x^l	変数
		|	$(\mathsf{let}\, x\ =\ e\, \mathsf{in}\, e)^l$	let 式
		|	$\mathsf{c}(e, \ldots, e)^l$	定数
Label	$l \in N$			ラベル
Value	$v \in Val$	::=	c	定数値
Environment	$env \in Env$	::=	.	空環境
		|	$[x \longmapsto v]env$	環境の拡張
Binder	$bind \in Binder$	::=	.	空束縛文脈
		|	$[x \longmapsto path]bind$	束縛文脈の拡張
Dirty Set	$ds \in 2^{Path}$			束縛した値が前回と異なるか
Path	$path \in Path$	::=	.	トップレベル
		|	$l : path$	実行文脈の拡張
History	$h \in History$	::=	.	空の履歴
		|	$[path \longmapsto (v,\ bind)]h$	履歴の拡張
Changes	$changes \in 2^{Label}$			プログラムの変更

図 3. データ構造

$$\frac{(v',\ bind') \doteq H(l : path) \quad labels(e) \cap changes = \emptyset \quad bind' \sqsubseteq bind \qquad dom(bind') \cap ds = \emptyset}{(e^l,\ env,\ bind,\ ds,\ path,\ h) \Downarrow (v',\ false,\ bind',\ combine(h,\ H,\ l : path))}\ (use\ history)$$

$bind' \sqsubseteq bind \doteq \forall x \in dom(bind').\ bind'(x) = bind(x)$
$lebels(e) \doteq e$のすべての部分式のラベルからなる集合

$combine(h,\ H,\ path) :\ Path \longrightarrow Value \times Bind$
$combine(h,\ H,\ path) \doteq h(p)\ if\ p \in dom(h)$
$combine(h,\ H,\ path) \doteq H(p)\ if\ p \in dom(H) \setminus D$

ただし、$D = \{\, p \in Path \mid path$は$path$の接尾辞$\,\}$

$$\frac{bind' \doteq bind \restriction_{\{x\}}}{(x^l,\ env,\ bind,\ ds,\ path,\ h) \Downarrow (env(x),\ true,\ bind',\ [l : path \longmapsto (env(x),\ bind')]h)}\ (var)$$

$$\frac{(e_i,\ env,\ bind,\ ds,\ l : path,\ h_{i-1}) \Downarrow (v_i,\ d_i,\ bind'_i\ h_i) \text{ for each } i \in \{1, \ldots, n\} \quad v \doteq \delta_{\mathsf{c}}(v_1, \ldots, v_n) \quad bind' \doteq \bigcup_{i=1}^{n} bind'_i \quad d \doteq \bigvee_{i=1}^{n} d_i}{(\mathsf{c}(e_1, \ldots, e_n)^l,\ env,\ bind,\ ds,\ path,\ h_0) \Downarrow (v,\ d,\ bind',\ [l : path \longmapsto (v,\ bind')]h_n)}\ (const)$$

$$\frac{\begin{array}{l}(e_1,\ env,\ bind,\ ds,\ h) \Downarrow (v_1,\ d_1,\ bind'1,\ h_1) \\ ds' \doteq (if\ d_1\ then\ ds\ else\ ds \cup \{l : path\}) \\ (e_2,\ [x \longmapsto v_1]env_1,\ [x \longmapsto l : path]bind,\ ds',\ l : path,\ h_1) \Downarrow (v_2,\ d_2,\ bind'2,\ h_2) \\ bind' \doteq bind'1 \cup bind'2\end{array}}{(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2,\ env,\ bind,\ ds,\ path,\ h) \Downarrow (v_2,\ d_2,\ bind',\ [l : path \longmapsto (v,\ bind')]h_2)}\ (let)$$

図 4. 評価規則

4.3. 通常実行の評価規則

安全性の性質を議論するために、通常実行の意味論を図 5 で定義する。通常実行の意味論は、差分実行を定めた評価規則から $(use\ history)$ を抜いたものとして定める。

$$\frac{bind' \doteq bind \restriction_{\{x\}}}{(x^l,\ env,\ bind,\ ds,\ path,\ h) \Downarrow_N (env(x),\ true,\ bind',\ [l : path \longmapsto (env(x),\ bind')]h)}\ (var)$$

$$\frac{\begin{array}{l}(e_i,\ env,\ bind,\ ds,\ l : path,\ h_{i-1}) \Downarrow_N (v_i,\ d_i,\ bind'_i\ h_i) \text{ for each } i \in \{1, \ldots, n\} \\ v \doteq \delta_{\mathsf{c}}(v_1, \ldots, v_n) \quad bind' \doteq \bigcup_{i=1}^{n} bind'_i \quad d \doteq \bigvee_{i=1}^{n} d_i\end{array}}{(\mathsf{c}(e_1, \ldots, e_n)^l,\ env,\ bind,\ ds,\ path,\ h_0) \Downarrow_N (v,\ d,\ bind',\ [l : path \longmapsto (v,\ bind')]h_n)}\ (const)$$

$$\frac{\begin{array}{l}(e_1^{l_1},\ env,\ bind,\ ds,\ h) \Downarrow_N (v_1,\ d_1,\ bind'1,\ h_1) \\ ds' \doteq (if\ d_1\ then\ ds\ else\ ds \cup \{l : path\}) \\ (e_2,\ [x \longmapsto v_1]env_1,\ [x \longmapsto l_1 : l : path]bind,\ ds',\ l : path,\ h_1) \Downarrow_N (v_2,\ d_2,\ bind'2,\ h_2) \\ bind' \doteq bind'1 \cup bind'2\end{array}}{(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2,\ env,\ bind,\ ds,\ path,\ h) \Downarrow_N (v_2,\ d_2,\ bind',\ [l : path \longmapsto (v,\ bind')]h_2)}\ (let)$$

図 5. 評価規則（通常実行）

4.4. 安全性

定理 1.

$(e, ., ., \emptyset, .) \Downarrow_N (v, \_, \_, \_) \implies \exists v'.\, H,\ changes \vDash (e, ., ., ., .) \Downarrow (v', \_, \_) \wedge v = v'$

H, $changes$ について、式 e_{prev} が存在して以下の条件を満たすことを仮定する。

- $(e_{prev}, ., ., \emptyset, .) \Downarrow_N (\_, \_, \_, H)$
- $e_{prev} \rightsquigarrow_{changes} e$

が成り立つ。ここで $e_{prev} \rightsquigarrow_{changes} e$ は以下のように定義される。

> e の任意の部分式 $e'l$ について、$l \notin changes$ ならば、e_{prev} の部分式にラベル l を持つ式 $e''l$ が存在して、
>
> - e' と e'' は同じ種類であり
> - 変数であれば変数名が一致し
> - let 式であれば変数名とそれぞれの部分式のラベルが一致し
> - 定数であれば定数のシンボルが一致し、全ての部分式について対応するもののラベル同士が一致する

補題 1.

式 H, $changes$, e_0, e_{prev} が存在して次の条件が成り立つことを仮定する。

- $(e_{prev},\ .,\ .,\ \emptyset,\ .) \Downarrow_N (\_,\ \_,\ \_,\ H)$
- $(e_0,\ .,\ .,\ \emptyset,\ .) \Downarrow_N (\_,\ \_,\ \_,\ h_0)$
- $e_{prev} \rightsquigarrow_{changes} e_0$

D_0 の任意の部分導出木 D をとる。D が $(e^l,\ path,\ env,\ bind,\ ds_N,\ h) \Downarrow_N (v_N,\ d_N,\ bind'_N,\ h'_N)$ を導出すると仮定する。$bind$, $path$ に対して条件 C を満たす任意の ds について、ある値 v, d, $bind'$, h' に対して差分実行の導出木 H, $changes \vDash (e^l,\ env,\ bind,\ ds,\ path,\ h) \Downarrow (v,\ d,\ bind',\ h')$ が存在して

1. $v_N = v$, $bind'_N = bind'$, $h'_N = h'$ かつ、
2. $d = false$ ならば $v = H(l : path).value$

が成り立つ。

ここで条件 C は以下のように定義する。

- C: $\forall path \in range(bind) \setminus ds.\ H(path).value = h(path).value$

証明. $(e^l,\ env,\ bind,\ path,\ h) \Downarrow_N (v,\ bind',\ h')$ の導出木の構造に関する帰納法で示す。導出木の規則で場合分けする。$(e_{prev},\ .,\ .,\ \emptyset,\ .) \Downarrow_N (\_,\ \_,\ \_,\ H)$ の導出木を D とする。

case (var)

仮定の導出木 D は以下のようになる。

$$\frac{bind' \doteq bind \restriction_{\{x\}}}{(x^l,\ env,\ bind,\ ds,\ path,\ h) \Downarrow_N (env(x),\ true,\ bind',\ [l : path \longmapsto (env(x),\ bind')]h)}\ (var)$$

subcase $(use\ history)$ の仮定が成り立たないとき

(var) 規則を用いて差分実行の導出木を以下のように構成できる。

$$\frac{bind' \doteq bind \restriction_{\{x\}}}{(x^l,\ env,\ bind,\ ds,\ path,\ h) \Downarrow (env(x),\ true,\ bind',\ [l : path \longmapsto (env(x),\ bind')]h)}\ (var)$$

したがって一つ目の命題が成り立つ。また $d = true$ なので 2 つ目の命題も成り立つ。

subcase $(use\ history)$ の仮定が成り立つとき

$(use\ history)$ を用いて差分実行の導出木を構成する。

$$\frac{\begin{array}{l}(v',\ bind') \doteq H(l : path) \quad labels(e) \cap changes = \emptyset \\ bind' \sqsubseteq bind \qquad dom(bind') \cap ds = \emptyset\end{array}}{(e^l,\ env,\ bind,\ ds,\ path,\ h) \Downarrow (v',\ false,\ bind',\ combine(h,\ H,\ l : path))}\ (use\ history)$$

- $v' = env(x)$
- $bind' = bind \restriction_{\{x\}}$
- $combine(h,\ H,\ l : path) = h'_N$

を示せば良い。

$l : path \in dom(H)$ なので e_{prev} の実行でも $(e^l_p,\ env_p,\ bind_p,\ ds_p,\ path,\ h_p)$ が実行されたことがわかる（実行されていないと仮定すると、H に履歴があることと矛盾するため）。

補題 2 から e^l は e_{prev} の部分式である。式のラベルの一意性から、$e^l = e^l_p$ が導ける。したがって D 中の e_p の導出木（D の部分木）は次のようにかける。

$$\frac{bind'_p \doteq bind_p \restriction_{\{x\}}}{(x^l,\ env_p,\ bind_p,\ ds_p,\ path,\ h_p) \Downarrow_N (env_p(x),\ true,\ bind'_p,\ [l : path \longmapsto (env_p(x),\ bind'_p)]h_p)}\ (var)$$

したがって $env(x) = env_p(x)$ を示せばよい。以下の等式変形から示せる。

$$\begin{aligned} env(x) &= h(bind(x)).value && (1)\\ &= h(bind_p(x)).value && (2)\\ &= H(bind_p(x)).value && (3)\\ &= env_p(x) && (4) \end{aligned}$$

等号 (1) − (4) が成り立つことを示す。

- (1),(4): 通常実行の仮定より
- (2): $bind_p \restriction_{\{x\}} = bind'_p = bind' \sqsubseteq bind$ より

$range(bind') \cap ds = \emptyset$ から $bind'(x)\ range(bind') \setminus ds$ が言える。したがって、仮定より $H(bind'(x)) = h(bind'(x))$ が成り立つ。$bind'(x) = bind_p(x)$ なので (3) がなりたつ。よって仮定より $env_p(x) = H(bind(x)) = h(bind(x)) = e$ がなりたつ。

したがって $env(x) = env_p(x)$ が成り立つ。

case (let)

仮定する通常実行の導出木は以下のようになる。

$$\frac{\begin{array}{l}(e_1^{l_1},\ env,\ bind,\ ds,\ l : path,\ h) \Downarrow_N (v_{1N},\ d_{1N},\ bind'1_N,\ h_{1N})\\ ds' \doteq (\mathsf{if}\ d_{1N}\ \mathsf{then}\ ds\ \mathsf{else}\ ds \cup \{l : path\})\\ (e_2,\ [x \longmapsto v_{1N}]env_1,\ [x \longmapsto l_1 : l : path]bind,\ ds',\ l : path,\ h_1) \Downarrow_N (v_{2N},\ d_{2N},\ bind'2_N,\ h_{2N})\\ bind' \doteq bind'1_N \cup bind'2_N\end{array}}{(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2,\ env,\ bind,\ ds,\ path,\ h) \Downarrow_N (v_{2N},\ d_2,\ bind',\ [l : path \longmapsto (v,\ bind')]h_{2N})}\ (let)$$

subcase $(use\,history)$ の仮定が満たされないとき

差分実行でも (let) 規則を用いる。以下のような差分実行の導出木を構成し、

1. $v_{2N} = v_2$
2. $bind'_N = bind'$
3. $[l : path \longmapsto (v,\ bind')]h_{2N} = [l : path \longmapsto (v,\ bind')]h_2$
4. $d_2 = false \implies v_2 = H(l : path)$

を示す。

$$\frac{\begin{array}{l}(e_1^{l_1},\ env,\ bind,\ ds,\ l : path,\ h) \Downarrow (v_1,\ d_1,\ bind'1,\ h_1)\\ ds' \doteq (\mathsf{if}\ d_1\ \mathsf{then}\ ds\ \mathsf{else}\ ds \cup \{l : path\})\\ (e_2,\ [x \longmapsto v_1]env_1,\ [x \longmapsto l_1 : l : path]bind,\ ds',\ l : path,\ h_1) \Downarrow (v_2,\ d_2,\ bind'2,\ h_2)\\ bind' \doteq bind'1 \cup bind'2\end{array}}{(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2,\ env,\ bind,\ ds,\ path,\ h) \Downarrow_N (v_2,\ d_2,\ bind',\ [l : path \longmapsto (v,\ bind')]h_2)}\ (let)$$

e_1 の通常実行の導出木に対して帰納法の仮定を用いることで、e_1 の差分実行の導出木が存在し、以下が成り立つことがわかる。

1. $v_{1N} = v_1$
2. ${bind_1}'_N = bind'_1$
3. $h_{1N} = h_1$

4. $d_1 = false \implies v_1 = H(l_1 : l : path)$

よって、帰納法の仮定から $e_2, [x \longmapsto v_1]env_1$ に対する差分実行の導出木が存在し、

1. $v_{2\,N} = v_2$
2. $bind_2{}'_N = bind'_2$
3. $h_{2\,N} = [l : path \longmapsto h_2$
4. $d_2 = false \implies v_2 = H(l_2 : l : path)$

が成り立つことがわかる。したがって

1. $bind'_N = bind_1{}'_N \cup bind_2{}'_N = bind'_1 \cup bind'_2 = bind'$
2. $[l : path \longmapsto (v, bind')]h_{2\,N} = [l : path \longmapsto (v, bind')]h_2$
3. $d_2 = false \implies v_2 = H(l_2 : l : path) = H(l : path)$

が成り立つ。

subcase $(use\,history)$ の仮定が満たされるとき

差分実行では $(use\,history)$ を用いる。

$$\frac{\begin{array}{l}(v', bind') \doteq H(l : path) \quad labels(e) \cap changes = \emptyset \\ bind' \sqsubseteq bind \qquad dom(bind') \cap ds = \emptyset\end{array}}{(e^l, env, bind, ds, path, h) \Downarrow (v', false, bind', combine(h, H, l : path))}\ (use\,history)$$

以下を示せば良い。

1. $v_{2\,N} = v'$
2. $bind'_N = bind'$
3. $[l : path \longmapsto (v, bind')]h_{2\,N} = combine(h, H, l : path)$
4. $v' = H(l : path)$

$H(l_1 : l : path), H(l_2 : l : path)$ の内容を明らかにした上で、今回の通常実行の e_1, e_2 に対する部分導出木について、帰納法の仮定を用いる。

$labels(e) \cap changes = \emptyset$ より、補題 2 より、$e_p = e$ が成り立つ。$l : path \in dom(H)$ なので e_{prev} の実行でも $(e^l_p, env_p, bind_p, ds_p, path, h_p)$ が実行されたことがわかる（実行されていないと仮定すると、H に履歴があることと矛盾するため)。その導出木は以下のようにかける。

$$\frac{\begin{array}{l}(e_1^{l_1}, env_p, bind_p, ds_p, l : path, h_p) \Downarrow_N (v_{p_1}, d_{p_1}, bind'_{p_1}, h_{p_1}) \\ ds'_p \doteq (\mathsf{if}\ d_{p_1}\ \mathsf{then}\ ds_p\ \mathsf{else}\ ds_p \cup \{l : path\}) \\ (e_2, [x \longmapsto v_{p_1}]env_p, [x \longmapsto l_1 : l : path]bind_p, ds'_p, l : path, h_{p_1}) \Downarrow_N (v_{p_2}, d_{p_2}, bind'_{p_2}, h_{p_2}) \\ bind'_p \doteq bind'_{p_1} \cup bind'_{p_2}\end{array}}{((\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)^l, env_p, bind_p, ds_p, path, h_p) \Downarrow_N (v_{p_2}, d_{p_2}, bind'_p, [l : path \longmapsto (v_{p_2}, bind'_p)]h_{p_2})}\ (let)$$

したがって、$H(l_1 : l : path) = (v_{p_1}, bind'_{p_1}), H(l_2 : l : path) = (v_{p_2}, bind'_{p_2})$ が成り立つ。

e_1 に対する通常実行の導出木について帰納法の仮定を用いることで $(e_1^{l_1}, env, bind, ds, l : path, h) \Downarrow (v_{1\,N}, d_1, b$ かつ $d_1 = false \implies v_{1\,N} = H(l_1 : l : path)$ が得られる。

e_2 に対する通常実行の導出木について帰納法の仮定を用いることで $(e_2^{l_2}, env, bind, ds, l : path, h) \Downarrow (v_{2\,N}, d_2, b$ かつ $d_2 = false \implies v_{2\,N} = H(l_2 : l : path)$ が得られる。

ところで、e_1, e_2 の差分実行は $(use\,history)$ 規則で導出される。そのため $(v1p, bind1p') = H(l_1 : l : path)$、

$(v2p,\ bind2p') = H(l_2 : l : path)$ である。したがって $v2p = v2n$、$bind1p' = bind1n'$、$bind2p' = bind2n'$ であり、…。

case $(const)$

通常実行の導出木は以下のようにかける。

$$\frac{\begin{array}{l}(e_i,\ env,\ bind,\ ds,\ l : path,\ h_{i-1}) \Downarrow (v_i,\ d_i,\ bind'_i\ h_i) \text{ for each } i \in \{1, \ldots, n\} \\ v \doteq \delta_{\mathsf{c}}(v_1, \ldots, v_n) \quad bind' \doteq \bigcup_{i=1}^{n} bind'_i \quad d \doteq \bigvee_{i=1}^{n} d_i\end{array}}{(\mathsf{c}(e_1, \ldots, e_n)^l,\ env,\ bind,\ ds,\ path,\ h_0) \Downarrow (v,\ d,\ bind',\ [l : path \longmapsto (v,\ bind')]h_n)}\ (const)$$

subcase $(use\ history)$ の仮定が成り立たないとき

差分実行でも $(const)$ を用いる。D に e と同等の状況の部分導出木が存在する。それぞれの部分木

補題 2. (プログラムの変更部分について)

以下が成り立つとき、e は e_{prev} の部分式である。

- $e_{prev} \rightsquigarrow_{changes} e_0$
- e^l は e_0 の部分式
- $labels(e) \cap changes = \emptyset$

証明. e^l のサイズに関する帰納法で示す。

e^l の種類で場合分けする。

case let 式のとき

$e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ とかける。$e_{prev} \rightsquigarrow_{changes} e_0$ の定義から、

- l をラベルにもつ e_{prev} の部分式 e'^l が存在して、e'^l は let 式である。
- $e' = \mathsf{let}\ x' = e_1'^{l'_1}\ \mathsf{in}\ e_2'^{l'_2}$ とすると
  - 変数名が一致するため $x = x'$
  - 部分式のラベルが一致するため $l_1 = l'_1,\ l_2 = l'_2$

が成り立つ。

$e_1,\ e_2$ はともに e の部分式であるから、$labels(e_1) \subset labels(e),\ labels(e_2) \subset labels(e)$ である。よって $e_1,\ e_2$ に帰納法の仮定を適用できて、$e_1,\ e_2$ が e_{prev} の部分式であることがわかる。

e_{prev} の部分式はラベルを一意にもつため $e_1 = e'_1,\ e_2 = e'_2$ である。したがって $e' = e$ が成り立つ。

case 変数のとき

$e_{prev} \rightsquigarrow_{changes} e_0$ の定義からあきらか。

case 定数のとき

let 式の場合と同様に示せる。

第 5 章

5. 関連研究

本研究ではプログラムの差分実行方式を形式化した。式の実行の差分を詳しく議論した点が特徴である。

5.1. プログラムの意味の差分

Grika らはプログラムの対に対して、それらの実行時の振る舞いの違いを表現したり、それを検証したりする手法を提案した [3, 4]。提案手法では、振る舞いの違いは correlating oracle によって表現、検証される。Correlating oracle は 2 つのプログラムを入力として受け取り、それらの実行状態の間に検証すべき関係が成り立つことを検証しながら実行を進める。2 つのプログラムの実行時の振る舞いの違いを、実行中の計算状況の単位で議論する点が本研究と類似している。

その一方で Grika らの研究は、2 つのプログラムの振る舞いの違いを特徴づけることに焦点を当て、振る舞いの違いはプログラマが与えるモデルを採用したのに対して、本研究では 2 つのプログラムの（部分的な）実行結果に焦点を当て、それらの違いを自動的に判定する手法を検討した。

5.2. 漸増計算

差分実行方式に関連した計算方式として、漸増計算 (incremental computation) がある。漸増計算ではプログラムの入力の一部が変更されるたびに、入力の変更部分に依存する部分だけを実行し出力を得ることで、計算時間を短縮する。入力データをプログラム、漸増計算をするプログラムをインタプリタと思えば、差分実行方式は漸増計算の一種であると解釈できる。

増分コンパイル (incremental compilation) は差分実行方式と同じくプログラムを入力とする漸増計算である。TypeScript 言語の JavaScript のトランスパイラも型検査での漸増計算をサポートする[2]ほか、OCaml 言語の静的解析ツールである Merlin は構文解析や型検査を漸増計算として実現している [2]。これらはプログラムを入力とする点では差分実行方式と類似するが、差分実行方式では実行にかかる時間を短縮することが目的である一方で、漸増コンパイルではプログラムのコンパイル時間を短縮することを目的とする。また、本研究で提案した差分実行方式は、式や値のレベルでの差分を扱うのに対して、これらの増分コンパイラや静的解析ツールでは、モジュールのようなより大きな単位での差分を扱う点が異なる。

[2] https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-4.html#faster-subsequent-builds-with-the---incremental-flag

第 6 章

6. 今後の課題と結論

本研究では差分実行方式とその正しさの性質を定式化し、正しさの証明を与えた。差分実行方式ではどのようなデータ構造が必要で、どのような不変条件を満たすことが必要化を明らかにした。

今後の課題としては、言語機能をより実用的にすることと、差分実行方式の最適化を与えることが考えられる。本研究で提案した形式化は言語機能が乏しく、差分実行方式を実装してライブプログラミング環境などで実用するための参考としては単純化しすぎている。クロージャや破壊的代入などをもつ言語に対しても形式化や安全性の証明を行うことは、実用的な処理系の実装の正しさを保証するために役立つだろう。形式化や証明を与える際には、本研究で与えた形式化や証明が参考になる。

既存の差分実行方式では、編集の後にプログラムをただ差分実行することだけを考えた。しかし、適切な前処理をすることで差分の有無の判定コストを下げることが可能だろう。そのような最適化を考えるために、本研究の形式化で差分の有無の判定に必要な計算量を見積もると良いだろう。

参考文献

[1] Sam Aaron. “Sonic Pi–performance in Education, Technology and Art.” *International Journal of Performance Arts and Digital Media* 12 (2). Taylor & Francis: 171 – 178. 2016.

[2] Frédéric Bour, Thomas Refis, and Gabriel Scherer. “Merlin: A Language Server for OCaml (Experience Report).” *Proc. ACM Program. Lang.* 2 (ICFP). Association for Computing Machinery, New York, NY, USA. Jul. 2018. doi:10.1145/3236798.

[3] Thibaut Girka. “Differential Program Semantics.” Theses, Université Paris Diderot. Jul. 2018. https://hal.inria.fr/tel-01890508.

[4] Thibaut Girka, David Mentré, and Yann Régis-Gianas. “Verifiable Semantic Difference Languages.” In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, 73 – 84. PPDP ’ 17. Association for Computing Machinery, New York, NY, USA. 2017. doi:10.1145/3131851.3131870.

[5] Akio Oka, Hidehiko Masuhara, and Tomoyuki Aotani. “Live, Synchronized, and Mental Map Preserving Visualization for Data Structure Programming.” In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 72 – 87. Onward! 2018. Association for Computing Machinery, New York, NY, USA. 2018. doi:10.1145/3276954.3276962.

[6] Patrick Rein, Stefan Lehmann, Toni Mattis, and Robert Hirschfeld. “How Live Are Live Programming Systems? Benchmarking the Response Times of Live Programming Environments.” In *Proceedings of the Programming Experience 2016 (PX/16) Workshop*, 1 – 8. PX/16. Association for Computing Machinery, New York, NY, USA. 2016. doi:10.1145/2984380.2984381.

[7] David Ungar, Henry Lieberman, and Christopher Fry. “Debugging and the Experience of Immediacy.” *Commun. ACM* 40 (4). Association for Computing Machinery, New York, NY, USA: 38 – 43. Apr. 1997. doi:10.1145/248448.248457.