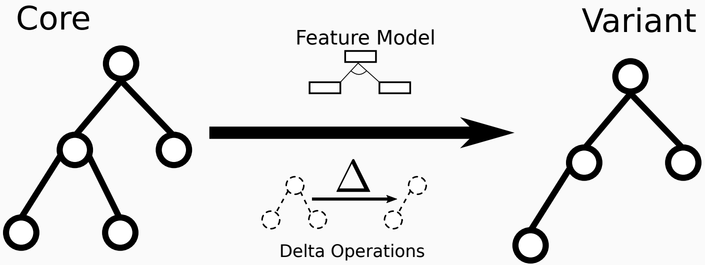
Deltas for Functional Programs with Algebraic Data Types

Ferruccio Damiani[2]
Eduard Kamburjan[1]
Michael Lienhardt[3]
Luca Paolini[2]

[1]University of Oslo, Norway
[2]University of Turin, Italy
[3]ONERA, France
SPLC, 08.09.23

Delta-Oriented Programming and Its Assumptions



Delta-Oriented Programming and Its Assumptions

```
class C { void m() { ... } }

delta d;
modifies class C; removes void m();
```

Delta-Oriented Programming and Its Assumptions

```
class C { void m() { ... } }


delta d;
modifies class C; removes void m();
```

- DOP has so far mostly been investigated for OO
- Family-based analyses, e.g., for typing, available
- DOP requires names to identify modification points
- Natural operation for FP: add/remove/modify named functions
- **How to handle pattern matching and algebraic data types?**

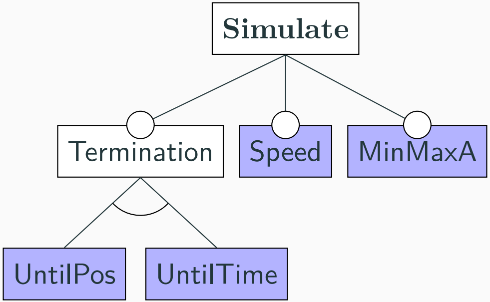
Challenges for DOP-FP

```
data Train = Train(Float pos, Float a, Float v, Spec spec);
data Aspect = Main(HaltAspect ha) | Pre(HaltAspect ha);
data HaltAspect = Stop | Pass;

def Train react(Train train, Aspect asp) =
    case asp {
        Main(Stop) => setA(train, -1.0);
        Main(Pass) => setA(train, 0.0);
        ... }};
```

- How to refer to specific branches?
- How to design a family-based type analysis?
- What intermediate properties are required?

Example: Train Simulation



- Different kinds of aspects (speed limiters) and trains (min/max acceleration)
- Different kinds of simulation (time-bound, position-bound)
- Connection of deltas and feature model omitted in the following

Language: Syntax by Example

```
def Train simulate(Pair<Train, List<Signal>> prs) =
  simulate(simulateStep(0.2, prs));
def Train react(Train train, Aspect asp) =
  case train as lbT {
  Train(pos, _, v, spec) =>
    case asp as lbA {
      Main(Stop) => setA(train, -1.0);
      Main(Pass) => setA(train, 0.0);
      Pre(Stop) => setA(train, -(v*v/20.0));
      Pre(Pass) => setA(train, ...); }};
```

- Standard functional constructs + optional labels

Deltas: Syntax by Example

```
delta dUntilTime;
modifies def Train simulate(Pair<Train, List<Signal>> prs)
 = case fst(prs) as lb {
     Train(_, _, v, _) => if(v <= 0.0) then fst(prs) else original(prs); };

delta dSpeed;
  modifies data HaltAspect { adds Speed(Float limit); }
  modifiesCase react { modifiesCase lbT {
  modifiesCase lbA {
    adds Main (Speed (target)) => setA(train, 0.0);
    adds Pre (Speed (target)) => setA(train, ...); }}};
```

- No modification of constructors (unclear order when modifying parameters)
- Parameters implicitly add functions (`Float limit(HaltAspect) = ...`)

Properties

We aim for the following properties

- Every generatable variant is type-safe (Type Safety)
- No generatable variant has incompatible patterns (Pattern Compatability)

To simplify analysis and provide guidance, we first establish the following properties:

Intermediate Properties

- Label consistency – use of labels is unambiguous
- Type-label-uniformity – constructs with multiple definitions are uniform
- No-useless-operations – no delta operation can be removed

Family-Based Analysis – Non-Variable Analyses

Label Consistency – Definition

- An SPL is *label consistent* if every path $f \ldots l$ has the same infix in all variants.
- Analysis idea: Check that all declarations of f have the same labels, slight complication for `original`

Type Uniformity – Definition

- An SPL is *type uniform* if every constructor/function is declared with the same signature in all variants

Partial Typing – Definition

- An SPL is *partially typed* if in every variant every use of a constructor/function is well-typed, or the constructor/function does not exist at all
- Analysis idea: Relies on prior analyses, overapproximation of the type table

Family-Based Analysis – Constraints

- If we have partial typing, we need to ensure that each constructor/function actually exists when added.
- Idea: collect constraint Φ for dependencies between use-sites and declarations
- In the end check $FM \wedge ACT \Rightarrow \Phi$ and establish type safety

Family-Based Analysis – Constraints

- If we have partial typing, we need to ensure that each constructor/function actually exists when added.
- Idea: collect constraint Φ for dependencies between use-sites and declarations
- In the end check $FM \wedge ACT \Rightarrow \Phi$ and establish type safety

Dependency Analysis by Example

- If a function `f` is used within a delta δ under path ρ, add constraint

$$Pre(\rho, \delta) \Rightarrow Pre(\mathtt{f})$$

- $Pre(\mathtt{f})$ encodes the activation condition of all deltas that add `f` such that no delta that removes it is activated
- $Pre(\rho, \delta)$ is analogous

Family-Based Analysis – Constraints

- If we have partial typing, we need to ensure that each constructor/function actually exists when added.
- Idea: collect constraint Φ for dependencies between use-sites and declarations
- In the end check $FM \wedge ACT \Rightarrow \Phi$ and establish type safety

$$\frac{\overline{\vdash \texttt{Dlt}_i : \Phi_i}}{\vdash \texttt{features}\ \overline{\mathsf{F}}\ \texttt{with}\ \Phi\ \texttt{Dlt}_1 \ \ldots\ \texttt{Dlt}_n\ \texttt{CK} : \bigwedge_{1\le i\le n} \Phi_i}\ \text{D:1}$$

$$\frac{\overline{d, T.\mathsf{K}_i \vdash v_1^i : \Phi_i^1 \quad \cdots \quad d, T.\mathsf{K}_i \vdash v_1^i : \Phi_{n_i}^1}}{d \vdash \texttt{data}\ T\langle a_1, \ldots a_n\rangle = \mathsf{K}_1(v_1^1, \ldots, v_{n_1}^1) \mid \cdots \mid \mathsf{K}_m(v_1^m, \ldots, v_{n_m}^m) : \bigwedge_{1\le i\le m} \bigwedge_{1\le j\le n_i} \Phi_j^i}\ \text{D:2}$$

$$\frac{d, \mathsf{f} \vdash e : \Phi \quad \overline{d, \mathsf{f} \vdash v_i : \Phi_i}}{d \vdash \texttt{def}\ \tau\ \mathsf{f}\langle a_1, \ldots a_n\rangle(\mathsf{x}_1 : v_1, \ldots, \mathsf{x}_m : v_m) = e : \Phi \wedge \bigwedge_{1\le i\le m} \Phi_i}\ \text{D:3} \qquad \frac{}{d, \rho \vdash a : \mathbf{true}}\ \text{D:4} \qquad \frac{d, \rho \vdash v : \Phi_1 \quad d, \rho \vdash \tau : \Phi_2}{d, \rho \vdash v \to \tau : \Phi_1 \wedge \Phi_2}\ \text{D:5}$$

$$\frac{\overline{d, \rho \vdash v_i : \Phi_i}}{d, \rho \vdash T\langle v_1, \ldots v_n\rangle : (Pre(\texttt{Spl}, \rho, d) \Rightarrow Pre(\texttt{Spl}, T)) \wedge \bigwedge_{1\le i\le n} \Phi_i}\ \text{D:6} \qquad \frac{}{d, \rho \vdash \mathsf{x} : \mathbf{true}}\ \text{D:7}$$

$$\frac{\overline{d, \rho \vdash e_i : \Phi_i} \quad \mathsf{K}\ \text{constructor of}\ T}{d, \rho \vdash \mathsf{K}(e_1, \ldots, e_n) : (Pre(\texttt{Spl}, \rho, d) \Rightarrow Pre(\texttt{Spl}, T.\mathsf{K})) \wedge \bigwedge_{1\le i\le n} \Phi_i}\ \text{D:8}$$

$$\frac{}{d, \rho \vdash \mathsf{f} : (Pre(\texttt{Spl}, \rho, d) \Rightarrow Pre(\texttt{Spl}, \mathsf{f}))}\ \text{D:9} \qquad \frac{d, \rho \vdash e_1 : \Phi_1 \quad d, \rho \vdash e_2 : \Phi_2}{d, \rho \vdash e_1\ e_2 : \Phi_1 \wedge \Phi_2}\ \text{D:10}$$

$$\frac{d, \rho \vdash e : \Phi \quad \overline{d, \rho.l.p \vdash e_i : \Phi_i} \quad \overline{d, \rho.l.p \vdash p_i : \Phi_i'}}{d, \rho \vdash \texttt{case}\ e\ \texttt{as}\ l\ \{\ p_1 \Rightarrow e_1\ \ldots p_n \Rightarrow e_n\ \} : \Phi \wedge \bigwedge_{1\le i\le n} (\Phi_i \wedge \Phi_i')}\ \text{D:11}$$

$$\frac{\overline{d \vdash \texttt{DtO}_i : \Phi_i} \quad \overline{d \vdash \texttt{FnO}_i : \Phi_i}}{\vdash \texttt{delta}d\ \texttt{DtO}_1\ \ldots\ \texttt{DtO}_n\ \texttt{FnO}_{n+1}\ \ldots\ \texttt{FnO}_m : \bigwedge_{1\le i\le m} \Phi_i}\ \text{D:12} \qquad \frac{d \vdash \texttt{DD} : \Phi}{d \vdash \texttt{adds}\ \texttt{DD} : \Phi}\ \text{D:13} \qquad \frac{}{d \vdash \texttt{removes}\ \texttt{data}\ T : \mathbf{true}}\ \text{D:14}$$

$$\frac{\overline{d, T \vdash \texttt{KAO}_i : \Phi_i}}{d \vdash \texttt{modifies}\ \texttt{data}\ T\ \{\ \overline{\texttt{KRO}}\ \texttt{KAO}_{n+1}\ \ldots\ \texttt{KAO}_m\ \} : \bigwedge_{1\le i\le m} \Phi_i}\ \text{D:15} \qquad \frac{\overline{d, T.\mathsf{K} \vdash v_i : \Phi_i}}{d, T \vdash \texttt{adds}\ \mathsf{K}(v_1, \ldots, v_n) : \bigwedge_{1\le i\le n} \Phi_i}\ \text{D:16}$$

$$\frac{d \vdash \texttt{FF} : \Phi}{d \vdash \texttt{adds}\ \texttt{FF} : \Phi}\ \text{D:17} \qquad \frac{}{d \vdash \texttt{removes}\ \texttt{def}\ f : \mathbf{true}}\ \text{D:18}$$

$$\frac{d \vdash \texttt{FF} : \Phi}{d \vdash \texttt{modifies}\ \texttt{FF} : \Phi}\ \text{D:19} \qquad \frac{\overline{d, \mathsf{f} \vdash \texttt{CaseO}_i : \Phi_i}}{d \vdash \texttt{modifiesCase}\ \mathsf{f}\ \{\ \texttt{CaseO}_1\ \ldots\ \texttt{CaseO}_n\} : \bigwedge_{1\le i\le n} \Phi_i}\ \text{D:20}$$

$$\frac{\rho_1 = \mathsf{f}.\rho.1 \quad \rho_2 = \Theta(\mathsf{f}, 1) \quad \overline{d, \rho_2 \vdash \texttt{BMO}_i : \Phi_i} \quad \overline{d, \rho_2 \vdash \texttt{BAO}_i : \Phi_i} \quad \overline{d, \rho.l \vdash \texttt{CaseO}_i : \Phi_i}}{d, \rho \vdash \texttt{modifiesCase}\ 1\ \{\ \overline{\texttt{BRO}}\ \texttt{BMO}_1\ \ldots\ \texttt{BMO}_m\ \texttt{BAO}_{m+1}\ \ldots\ \texttt{BAO}_o\ \texttt{CaseO}_{o+1}\ \ldots\ \texttt{CaseO}_p\ \} : \bigwedge_{1\le i\le p} \Phi_i}\ \text{D:21}$$

$$\frac{d, \rho.p \vdash e : \Phi}{d, \rho \vdash \texttt{modifies}\ p \Rightarrow e : \Phi}\ \text{D:22} \qquad \frac{d, \rho.p \vdash e : \Phi}{d, \rho \vdash \texttt{adds}\ p \Rightarrow e : \Phi}\ \text{D:23}$$

Properties

- Besides type safety, we show two more properties
- Both are based on different constraints and dependencies, but same scheme

Pattern compatibility

- No generable variant of an SPL has overlapping patterns in a case-expression.
- Constraint express dependencies between variables in patterns and constructors

Applicability consistency

- All variants of an SPL can be generated. No errors occur during flattening.
- Constraint express dependencies between remove operations and add operations

Conclusion

Conclusion

Integration with OO

- Elegantly integrates with OO for multi-paradigm languages
- Syntactically and semantically analogous operations
- Dependency analysis generalizes to FP for new properties!
- On-going implementation in ABS

Conclusion

Integration with OO

- Elegantly integrates with OO for multi-paradigm languages
- Syntactically and semantically analogous operations
- Dependency analysis generalizes to FP for new properties!
- On-going implementation in ABS

Summary

- Extension of DOP to FP
- Family-Based Analyses for patterns and types
- Future Work: Generalization to paradigm-independent framework

Conclusion

Integration with OO

- Elegantly integrates with OO for multi-paradigm languages
- Syntactically and semantically analogous operations
- Dependency analysis generalizes to FP for new properties!
- On-going implementation in ABS

Summary

- Extension of DOP to FP
- Family-Based Analyses for patterns and types
- Future Work: Generalization to paradigm-independent framework

Thank you for your attention