Deltas for Functional Programs with Algebraic Data Types

Ferruccio Damiani
ferruccio.damiani@unito.it
University of Turin, Turin
Italy

Eduard Kamburjan
eduard@ifi.uio.no
University of Oslo, Oslo
Norway

Michael Lienhardt
michael.lienhardt@onera.fr
ONERA, Paleseau
France

Luca Paolini
luca.paolini@unito.it
University of Turin, Turin
Italy

ABSTRACT

The development of feature-oriented programming (FOP) and of (its generalization) delta-oriented programming (DOP) has focused primarily on SPLs of class-based object oriented programs. In this paper, we introduce delta-oriented SPLs of functional programs with algebraic data types (ADTs). To pave the way towards SPLs of multi-paradigm programs, we tailor our presentation to the functional sublanguage of the multi-paradigm modeling language ABS, which already features DOP support for its class-based object-oriented sublanguage. Our main contributions are: (i) we motivate and illustrate our proposal by an example from an industrial modeling scenario; (ii) we formalize delta-oriented SPLs for functional programs with ADTs in terms of a foundational calculus; (iii) we define family-based analyses to check whether an SPL satisfies certain well-formedness conditions and whether all variants can be generated and are well-typed; and (iv) we briefly outline how, in the context of the toolchain of ABS, the proposed delta-oriented constructs and analyses for functional programs can be integrated with their counterparts for object-oriented programs.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; *Functional languages.*

KEYWORDS

Delta-oriented programming, Family-based analysis, Functional programming, Language design, Type checking

ACM Reference Format:
Ferruccio Damiani, Eduard Kamburjan, Michael Lienhardt, and Luca Paolini. 2023. Deltas for Functional Programs with Algebraic Data Types. In *27th ACM International Systems and Software Product Line Conference - Volume A (SPLC '23), August 28-September 1, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3579027.3608977

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SPLC '23, August 28-September 1, 2023, Tokyo, Japan
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0091-0/23/08…$15.00
https://doi.org/10.1145/3579027.3608977

1 INTRODUCTION

The development of *feature-oriented programming (FOP)* [1] and of (its generalization) *delta-oriented programming (DOP)* [19] has focused primarily on SPLs of class-based object oriented programs. In particular, to the best of our knowledge, no FOP/DOP approach for SPLs of functional programs can be found in the literature. In this paper, we argue that support for delta-oriented SPLs of functional programs with *algebraic data types (ADTs)* would be useful, and we propose and formalize it for the functional sublanguage of the multi-paradigm modeling language ABS [14]. We choose ABS because it already features DOP support for its class-based object-oriented sublanguage [5, 7]; so providing DOP support for its functional sublanguage would pave the way to investigate SPLs of multi-paradigm programs. Notably, ABS was successfully used in industrial case studies [15, 17, 24] and the biggest of these case studies [15] comprises both functional and class-based object-oriented code, but was not able to properly implement variability for its functional code due to a lack of support in ABS.

A functional program consists of a set of ADT definitions and a set of function definitions. A delta-oriented SPL [19] consists of a feature model, a base program (usually representing a variant), a set of deltas, and configuration knowledge. Each delta consists of a set of delta-operations, specifying changes to the base program. Configuration knowledge specifies an order of application between the deltas, and a set of activation conditions over features (exactly one for each delta) specifying which deltas to apply to the base program to generate the variant associated with the selected product.

Inspired by the formulation of DOP for SPLs of class-based object-oriented programs [19], it is straightforward to identify the following delta operations on functional programs: (i) add, remove, or modify (by adding or removing a constructor) an ADT definition; and (ii) add, remove, or modify (while possibly calling its original version) a function definition.

However, our case study (outlined in Sect. 2) shows that the presence of ADTs poses an interesting challenge: more fine-grained operations on function definitions are required to complement the addition/removal of a constructor in an ADT $\texttt{T}$ by adding/removing accordingly the corresponding branch of the case-expressions $\texttt{case}$ $e_0 \{ p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n \}$ ($n \geq 1$) where e_0 has type $\texttt{T}$. Namely, we need operations to add/remove a branch $p_i \Rightarrow e_i$ (where p_i is a pattern matching expression and e_i is the expression to be evaluated when the match succeeds) of a case-expression, and also to modify it (by replacing its right-hand side e_i). In order to be able to identify

the case-expression to be modified we extend the syntax of case-expressions with a label (which cannot occur twice inside a same function definition). Then, within a given case-expression, each branch $p_i \Rightarrow e_i$ can be identified by its left-hand side p_i. This solves the problem of identifying the case-expression to be modified and the branch to be removed or modified. The last issue to be solved concerns the place where a delta should add a branch to a case-expression: (i) a case-expression is evaluated by considering its branches in the order they appear; and (ii) unfortunately (due to the possible long sequence of add, remove and modify operations used to generate each variant) it seem almost impossible to devise a usable DOP construct to control the order in which branches occur in each variant. To rule out this issue we enforce the following syntactic restrictions on case expressions:

R1 Each case expression may contain at most one *default* branch, i.e., a branch of the form $x \Rightarrow e$ (for any variable x) which matches with every expression e_0 and is at the last place in the list of branches.

R2 All the branches but the optional default branch must be mutually exclusive – so their order becomes immaterial!

The main contributions of this paper are:

(1) we motivate the need of variability support for SPLs of functional programs with ADTs by an ABS example from an industrial modeling scenario (in Sect. 2);
(2) we devise and formalize a core functional programming language with ADTs that is suitable for DOP (in Sect. 3) and we provide a formal foundation for delta-oriented SPLs of functional programs with ADTs (in Sect. 4);
(3) we define and formalize properties and family-based analyses to check whether an SPL satisfies certain well-formedness conditions and all variants can be generated and are well-typed (in Sects. 5 and 6); and
(4) we briefly outline how, in the context of the toolchain of the ABS multi-paradigm modeling language, the proposed delta-oriented constructs and analyses for functional programs can be integrated with their counterparts for class-based object-oriented programs (in Sect. 7).

2 RUNNING EXAMPLE

SPLs of railway operations [15] and of simulators [9] have been proven useful in practice. We use an SPL of (sketched) simulators of train operations to illustrate our approach, in particular the connection between data and functions. Each variant of the SPL simulates the operations of a single train on a line between two stations, where it must adhere to physical laws (for acceleration), as well as to the operations rulebooks to react to the signals along the line. We introduce the Base Program first, then the SPL.

Base Program: a (sketched) Train Simulator. Fig. 1 gives the code of the Base Program, which is the variant of the SPL corresponding to the empty configuration. It defines the basic data structures. Datatype `Train` contains the dynamic aspects of a train (position `pos`, acceleration `a`, speed `v`) and a link to its static specification `spec`.[1] This static specification, defined by datatype `Spec`, contains only the

```
1  data Train = Train(Float pos, Float a, Float v, Spec spec);
2  data Spec = Spec(Float maxV);
3  data Signal = Signal(Float pos, Aspect asp)
4  data Aspect = Main(HaltAspect ha) | Pre(HaltAspect ha);
5  data HaltAspect = Stop | Pass;
6
7  def Spec getSpec(Train train) = case train as lb {
8    Train(_, _, _, spec) => spec;}
9  def Train setA(Train train, Float newA) = case train as lb {
10   Train(pos, _, v, spec) => Train(pos, newA, v, spec);}
11 def Float maxV(Train t) = case getSpec(t) as lb {
12   Spec(x) => x;}
13 def Train simulate(Pair<Train, List<Signal>> prs) =
14   simulate(simulateStep(0.2, prs));
15 def Train react(Train train, Aspect asp) =
16  case train as lbT {
17   Train(pos, _, v, spec) =>
18    case asp as lbA {
19     Main(Stop) => setA(train, -1.0); //emergency break
20     Main(Pass) => setA(train, 0.0);
21     Pre(Stop) => setA(train, -(v*v/20.0));
22     Pre(Pass) => setA(train, ...); }};
23 def Pair<Train, List<Signal>> simulateStep(Float step,
24         Pair<Train,List<Signal>> prs)=...
25 def Unit main() = ...
```

Figure 1: Base Program: a (sketched) train simulator.

maximum possible speed of the train, `maxV`. A signal is represented by the datatype `Signal` that has a position along the track (`pos`) and has a certain aspect (`asp`). The signal aspect, represented by the datatype `Aspect` marks the signal as a `Main` signal or a `Pre` signal. A main signal marks the position where a communicated aspect must be observed (e.g., to stop), while the pre signal announces the main signal (so the train has time to react). Additionally, the aspect itself, represented by the datatype `HaltAspect` is to either `Stop` at the main signal, or to `Pass` it. Operating on these data, we have two helper functions to update the acceleration of a train (`setA`) and to unwrap its maximum velocity (`maxV`), respectively.

The main simulation loop is given by the function `simulate`, which takes as input a train, a list of events (i.e., signals along the track) and simulates for 0.2 steps by calling `simulateStep` – in the base variant, the loop does not terminate and must be handled outside the program (e.g., in an interactive environment for the developer). Function `simulateStep` returns a pair consisting of a train and list of events.[2] It takes care of physics and detection of events, and performs the physical movement, as well as triggers events if necessary. Function `react` models the reaction of a train to a concrete event. We omit the body of some functions that are not variable (such as `simulateStep` and `main`), as well as some complex expressions (like the second argument of the call of function `setA` in line 23).

An SPL of (sketched) Train Simulators. The feature model, which specifies 12 products, is pictured in Fig. 2 as a feature diagram. As usual in DOP [19], in the logical formulation of the feature model, shown in Fig. 3 (top), only the concrete features (in blue in Fig. 2) are considered. We consider variability of the simulator infrastructure itself via the `Termination` feature, where the simulation stops once the train is not moving (`UntilPos`) or a certain amount of time has passed (`UntilTime`) and variability in data and functions operating on them: (i) Modeling more static information, such as maximum and minimum acceleration (`MinMaxA`), which the simulator accesses to break correctly – note that the acceleration of the emergency break

[1]For sake of readability, we add a label to each ADT constructor argument (as allowed by the syntax of ABS).

[2]In ABS, the standard datatypes `Pair<X,Y>` and `List<Z>` are built-in.

```
1  // Feature model
2  features UntilPos, UntilTime, Speed, MinMaxA
3       with  !(UntilPos && UntilTime)
4  // Configuration Knowledge: Activation conditions
5  delta dNoPass when Speed;
6  delta dUntilPos when UntilPos;
7  delta dUntilTime when UntilTime;
8  delta dSpeed when Speed;
9  delta dMinMaxA when MinMaxA;
10 // Configuration Knowledge: Application order
11 dNoPass  <  dUntilPos dUntilTime  <  dSpeed dMinMaxA;
```

Figure 3: Feature Model and Configuration Knowledge.

in `react` is hard coded so far; and (*ii*) modeling more procedural variants, such as speed limiters which transmit the command to pass the signal with a certain speed (`Speed`).

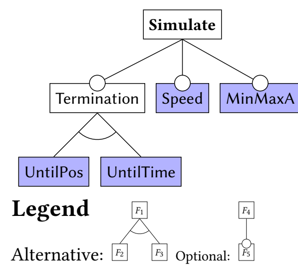

Figure 2: Feature Model.

For the variability of termination conditions, shown by deltas `dUntilPos` and `dUntilTime` in Fig. 4, only the `simulate` function needs to be modified. The delta `dUntilPos` modifies the function by replacing it with a new variant that checks whether the position of the train is simulated past 2000m (`snd` returns the 2nd element of a pair). If this is the case, then the final state of the train is returned. Otherwise, the **`original`** version of the function, i.e., before the delta modified the function, is called. The delta `dUntilTime` is analogous, but checks whether the train has come to halt. Only one such condition can be selected.

The addition of the speed limiter, done by delta `dSpeed`, requires to add a new constructor to the datatype `HaltAspect`, so the **`case`** expression on `HaltAspect` in function `react` needs to be adjusted. This is done by a **`modifiesCase`** on the function `react`, which contains a **`modifiesCase`** on the case expression labeled `lbT`, which in turn contains a **`modifiesCase`** on the case expression labeled `lbA`, which adds two new branches.

If one has speed limiters, one may under certain conditions, model a `Pass` aspect as a speed limiter with the corresponding maximum speed. The `Pass` aspect can, thus, be then removed. This is done by the `dNoPass` delta.

The addition of new static information to the train requires the biggest changes, which are expressed by delta `dMinMaxA` in Fig. 5. It adds a new constructor `ExtendSpec` to the datatype `Spec`, and modifies several functions. Function `maxV` shows the interactions of data and functions: it must be extended to handle the new constructor. The new functions `minA` and `maxA` are just for unwrapping, and function `setA` is modified to take the new limits for acceleration into account – note that the old constructor is handled with the default branch. Finally, in function `react` we modify a branch by replacing it with the minimal allowed acceleration for emergency breaking.

Configuration knowledge, shown in Fig. 3 (bottom), connects the deltas to the features. Delta `dNoPass` is activated by feature `Speed`, delta `dUntilPos` is activated by feature `UntilPos`, etc. The application order specifies that (whenever activated) `dNoPass` must be applied

```
1  delta dNoPass;
2  modifies data HaltAspect { removes Pass; }
3  modifiesCase react { modifiesCase lbT {
4   modifiesCase lbA {
5    removes Main (Pass);
6    removes Pre (Pass); }}};
7
8  delta dUntilPos;
9  modifies def Train simulate(Pair<Train, List<Signal>> prs)
10  = case fst(prs) as lb {
11     Train(pos, _, _, _) =>
12       if(pos >= 2000) then fst(prs)
13       else original(prs); };
14
15 delta dUntilTime;
16 modifies def Train simulate(Pair<Train, List<Signal>> prs)
17  = case fst(prs) as lb {
18     Train(_, _, v, _) =>
19       if(v <= 0.0) then fst(prs)
20       else original(prs); };
21
22 delta dSpeed;
23 modifies data HaltAspect { adds Speed(Float limit); }
24 modifiesCase react { modifiesCase lbT {
25  modifiesCase lbA {
26   adds Main (Speed (target)) => setA(train, 0.0);
27   adds Pre (Speed (target)) => setA(train, ...); }}};
```

Figure 4: Termination conditions (deltas `dUntilPos` and `dUntilTime`), speed limit aspect (delta `dSpeed`), removal of passing aspect (delta `dNoPass`).

```
1  delta dMinMixA;
2  modifies data Spec {
3   adds ExtendSpec(Float maxV, Float minA, Float maxA);}
4   modifiesCase maxV { modifiesCase lb {
5     adds ExtendSpec(x, _, _) => x; }}
6   adds def Float minA(Train train) =
7     case getSpec(train) as lb {
8      ExtendSpec(_, x, _) => x; _ => -1.0;}
9    ... // analogous for maxA
10  modifies def Train setA(Train train, Float newA)
11    = if(newA >= maxA(train))
12        then Train(.., maxA(train), ..)
13      else if(newA <= minA(train))
14        then Train(.., minA(train), ..)
15      else original(train, newA);
16  modifiesCase react { modifiesCase lbT {
17    modifiesCase lbA {
18     modifies Main(Stop) => setA(train, minA(train)); }}}
```

Figure 5: Adding a more complex static structure to trains.

before `dUntilPos` and `dUntilTime` which, in turn, must be applied before `dMinMaxA` and `dSpeed`. When features `UntilPos`, `Speed` and `MinMaxA` are selected, the variant in Fig. 6 is generated.

If the developer would have inserted by mistake the line of code

```
removes Main (Speed (target))
```

between lines 4 and 5 in Fig. 4, then the generation of the variant shown in Fig. 6 would fail, because the application of delta `dNoPass` would fail, because the new **`removes`**-operation would fail (there would be no branch "`Main (Speed (target)) => ...`" to be removed, because the delta `dNoPass` is applied first.

If the developer would have forgot to write line 5 in Fig. 4, then the variant shown in Fig. 6 would contain, between lines 36 and 37, the code in line 36 of Fig. 1. Since, in the given variant the data type `HaltAspect` has no constructor `Pass`, this would be a type error.

```
1  data Train = Train(Float pos, Float a, Float v, Spec spec);
2  data Spec = Spec(maxV:Float)
3            | ExtendSpec(Float maxV, Float minA, Float maxA);
4  data Signal = Signal(Float pos, Aspect asp)
5  data Aspect = Main(HaltAspect ha) | Pre(HaltAspect ha);
6  data HaltAspect = Stop | Speed (limit:Float);
7
8  def Train setA(Train train, Float newA)
9   = if(newA >= maxA(train))
10       then Train(.., maxA(train), ..)
11     else if(newA <= minA(train))
12       then Train(.., minA(train), ..)
13     else setA_core(train, newA);
14 def Train setA_core(Train train, Float newA)
15  = case train {
16    Train(pos, _, v, spec) => Train(pos, newA, v, spec) };
17 def Float maxV(Train train) =
18  case getSpec(train) as lb {
19    ExtendSpec(x,_,_) => x;
20    Spec(x) => x;}
21 adds def Float minA(Train train) =
22  case getSpec(train) as lb {
23    ExtendSpec(_, x, _) => x; _ => -1.0;}
24 // analgous for maxA
25 def Train simulate_core(Train train, List<Signal> evs)
26  = simulate(simulateStep(train, 0.2, evs));
27 def Train simulate(Train train, List<Signal> evs)
28  = case train {
29      Train(pos, _, _, _) =>
30        if(pos >= 2000) then train
31        else simulate_core(train, evs); };
32 def Train react(Train train, Aspect asp) =
33  case train as lbT {
34    Train(pos, _, v, spec) =>
35     case asp as reactL {
36      Main(Stop) => setA(train, minA(train));
37      Pre(Stop) => setA(train, -(v(train)*v(train)/20.0));
38      Main (Speed (target)) => setA(train, ...);
39      Pre (Speed (target)) = train;};
40 def Pair<Train, List<Signal>> simulateStep(...)=...
41 def Unit main() = ...
```

Figure 6: Variant for product `{UntilPos, Speed, MinMaxA}`.

3 F²ABS: A CORE FUNCTIONAL LANGUAGE

F²ABS (Featherweight Functional ABS) is a core language, with an ABS-like vanilla syntax, which formalizes explicitly typed higher-order functions and ADTs.[3] Following [13], $\overline{X}$ denotes a possibly empty finite sequence of elements X. In accordance with [11], we optionally enrich this notation by annotating its righthand side with the letter indexing the elements of the sequence. For instance, if $1 \leq k \leq n$ then $[\ \overline{a_k \mapsto v_k}^{k}\ ]\tau_i$ is a shorthand for $[a_1 \mapsto v_1, \ldots, a_n \mapsto v_n]\tau_i$. Moreover, we use $\equiv$ to denote syntactic equality.

Syntax. The syntax of our core language is given in Figure 7. Programs are formed by a sequence of (possibly mutually recursive) ADT definitions followed by a sequence of (possibly mutually recursive) function definitions.

Type variables are ranged over by a, b. We write T to denote names of ADTs. Data constructors are ranged over by K. Function names are ranged over by f,g,h and formal parameter names are ranged over by x,y,z. ADTs are defined by enumerating the data constructors K and the types of their arguments v (that may contain occurrences of the type parameters in $\overline{a}$).

$\texttt{Prg} ::= \overline{\texttt{DD}}\ \overline{\texttt{FF}}$	**Program**
$\texttt{DD} ::= \textbf{data}\ T\langle\overline{a}\rangle = \mathsf{K}_1(\overline{v_1}) \mid \cdots \mid \mathsf{K}_m(\overline{v_m})$	**ADT Definition**
$\texttt{FF} ::= \textbf{def}\ \tau\ \mathsf{f}\langle\overline{a}\rangle(\overline{v\ \mathsf{x}}) = e$	**Function Definition**
$\tau, v ::= a \mid v \rightarrow \tau \mid T\langle\overline{\tau}\rangle$	**Type**
$e ::= \mathsf{x} \mid \mathsf{K}(\overline{e}) \mid \mathsf{f} \mid e\ e \mid \textbf{case}\ e\ \textbf{as}\ l\ \{ p \Rightarrow e\ \overline{\mid p \Rightarrow e}\ \}$	**Expression**
$p ::= \mathsf{x} \mid \mathsf{K}(\overline{p})$	**Pattern**

Figure 7: A vanilla syntax for F²ABS.

An expression is either: a variable; a data constructor applied to all its arguments; a function name; a function application; or a case expression, comprising the expression to be matched, the label l, and the branches (viz. pairs "pattern ⇒ expression").

In each branch of a case, the variables occurring in the left-hand pattern bound the occurrences in the right-hand expression. Patterns can be nested, so a pattern can contains sub-patterns involving constructor of another ADT. In a pattern, variables must be linearly used, hence each variable can occur at most once in a pattern. We assume that the patterns in the branches of each case are mutually exclusive, except for the optional ending default branch of the form x⇒e, for some variable x.

Given a 0-ary data constructor K, we will sometimes write K as short for K(). Since data constructs must be always applied to all their arguments, no confusion may arise. Similarly, given a datatype T with no type parameters, we will sometimes write T as short for $\mathtt{T}\langle\rangle$.

Following [14], we do not include neither anonymous functions, nor **let**-expressions, nor **if**-expressions in our core language: they are syntactic sugar. We use Unit, Bool, Pair, List, Int, Float to denote the built-in datatypes:

```
data Unit = Unit                    data Bool = True | False
data Pair⟨a1, a2⟩ = Pair(a1, a2)    data List⟨a⟩ = Nil | Cons(a,List⟨a⟩)
data Int = -N | ... | -2 | -1 | 0 | 1 | 2 | ... | N
data Float = -Inf | ... | -2.01 | ... | 0.0 | ... | Inf | NaN
```

where we assume N be a natural number. So, the code given in Fig. 1 is an example of F²ABS program where, for readability sake, we have used the anonymous variable _ in patterns and the terminator ";" as in the ABS syntax.

Typing. The syntax of polymorphic types, environments, and substitutions is given in Figure 9. If $\overline{v}$ denotes $v_1, \ldots, v_n$ then, we write $\overline{v} \rightarrow \tau$ as a shortening for $v_1 \rightarrow \cdots \rightarrow v_n \rightarrow \tau$ $(n \geq 0)$. An environment maps expression variables to types, ADT constructor names to polymorphic types of the form $\forall\overline{a}.\overline{v} \rightarrow \mathrm{T}\langle\overline{a}\rangle$, and funcion names to polymorhic types of the form $\forall\overline{a}.\overline{v} \rightarrow \tau$. As usual, the operation of environment concatenation (denoted simply by a comma) is well-defined if and only if, either no name assignment clashes or same names are assigned to same types.

The typing rules for the core language are given in Figure 8 – they are based on the formalization of Haskell 98 proposed in [11, Fig.3]. Rule PRG is applied to conclude that the program $\texttt{Prg} = \overline{\texttt{DD}}\ \overline{\texttt{FF}}$ is well-typed. Its premise $\vdash_{\mathrm{PD}} \overline{\texttt{DD}} \rightsquigarrow \Gamma_0$ can be concluded by the rule PD that parses the list $\overline{\texttt{DD}}$ of datatypes declarations: it collects in

[3] The functional sublanguage of ABS [14] does not support higher-order functions. Instead, although it has an ABS-like syntax, F²ABS is essentially an higher-order explicitly typed lambda calculus with ADTs – like, e.g., those presented in [11]. So, the techniques developed in this paper represent a foundation for delta-oriented SPLs of higher-order explicitly typed functional programs.

Program Typing

$$\frac{\texttt{Prg} = \overline{\texttt{DD}}\,\overline{\texttt{FF}} \qquad \vdash_{\text{pD}} \overline{\texttt{DD}} \rightsquigarrow \Gamma_0 \qquad \vdash_{\text{pF}} \overline{\texttt{FF}} \rightsquigarrow \Gamma_1 \qquad \overline{\texttt{DD}} \propto \Gamma_0 \qquad \Gamma_0, \Gamma_1 \vdash_{\text{FF}} \overline{\texttt{FF}}}{\vdash \texttt{Prg}}\ \textsc{Prg}$$

Populating Environment

$$\frac{\texttt{FF}_i \ \equiv\ \texttt{def}\ \tau_i\ \mathsf{f}_i\langle \overline{a^i_k}^k \rangle (\overline{v^i_j\ \mathsf{x}^i_j}^j) = e_i \qquad \text{where } 1 \le i \le n}{\vdash_{\text{pF}} \texttt{FF}_1 \ldots \texttt{FF}_n \rightsquigarrow \mathsf{f}_1 : \forall \overline{a^1_k}^k . \overline{v^1_j}^j \to \tau_1, \ldots, \mathsf{f}_n : \forall \overline{a^n_k}^k . \overline{v^n_j}^j \to \tau_n}\ \text{pF}$$

$$\frac{\texttt{DD}_i \ \equiv\ \texttt{data}\ T_i\langle \overline{a^i} \rangle = \mathsf{K}^i_1(\overline{v^{i,1}}) \mid \cdots \mid \mathsf{K}^i_{m_i}(\overline{v^{i,m_i}}) \qquad \text{where } 1 \le i \le n}{\vdash_{\text{pD}} \texttt{DD}_1 \ldots \texttt{DD}_n \rightsquigarrow \left\{ \begin{array}{l} \mathsf{K}^1_1 : \forall \overline{a^1} . \overline{v^{1,1}} \to T_1\langle \overline{a^1} \rangle, \ldots, \mathsf{K}^1_{m_1} : \forall \overline{a^1} . \overline{v^{1,m_1}} \to T_1\langle \overline{a^1} \rangle, \\ \quad \cdots, \quad \cdots, \quad \cdots, \\ \mathsf{K}^n_1 : \forall \overline{a^n} . \overline{v^{n,1}} \to T_n\langle \overline{a^n} \rangle, \ldots, \mathsf{K}^n_{m_n} : \forall \overline{a^n} . \overline{v^{n,1}} \to T_n\langle \overline{a^n} \rangle \end{array} \right.}\ \text{pD}$$

Function Typing

$$\frac{\texttt{FF}_i \ \equiv\ \texttt{def}\ \tau_i\ \mathsf{f}_i\langle \overline{a^i} \rangle (\overline{v^i\ \mathsf{x}^i}) = e_i; \qquad \Gamma, \overline{\mathsf{x}^i : v^i} \vdash_{\text{E}} e_i : \tau_i \qquad \text{where } 1 \le i \le n}{\Gamma \vdash_{\text{FF}} \texttt{FF}_1 \ldots \texttt{FF}_n}\ \text{FF}$$

Expression Typing

$$\frac{(\mathsf{x} : \tau) \in \Gamma}{\Gamma \vdash_{\text{E}} \mathsf{x} : \tau}\ \textsc{Var} \qquad \frac{(\mathsf{f} : \forall \overline{a_i}^i . \tau) \in \Gamma}{\Gamma \vdash_{\text{E}} \mathsf{f} : [\ \overline{a_i \mapsto v_i}^i\ ]\tau}\ \textsc{Fn} \qquad \frac{(\mathsf{K} : \forall \overline{a_k}^k . \overline{\tau_i}^i \to T\,\overline{a_k}^k) \in \Gamma \qquad \overline{\Gamma \vdash_{\text{E}} e_i : [\ \overline{a_k \mapsto v_k}^k\ ]\tau_i}^i}{\Gamma \vdash_{\text{E}} \mathsf{K}(\overline{e_i}^i) : T\langle \overline{v_k}^k \rangle}\ \textsc{Con}$$

$$\frac{\Gamma \vdash_{\text{E}} e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\text{E}} e_2 : \tau_1}{\Gamma \vdash_{\text{E}} e_1\ e_2 : \tau_2}\ \textsc{App} \qquad \frac{\Gamma \vdash_{\text{E}} e : v \qquad \overline{\Gamma \vdash_{\text{P}} p_i : v \rightsquigarrow \Gamma'_i \qquad \Gamma'_i \vdash_{\text{E}} e_i : \tau}^i}{\Gamma \vdash_{\text{E}} \texttt{case}\ e\ \texttt{as}\ l\ \{\ \overline{p_i \Rightarrow e_i}^i\ \} : \tau}\ \textsc{Case}$$

Pattern Typing

$$\frac{}{\Gamma \vdash_{\text{P}} \mathsf{x} : \tau \rightsquigarrow \Gamma, \mathsf{x} : \tau}\ \textsc{PatVar} \qquad \frac{(\mathsf{K} : \forall \overline{a_i}^i . v_1 \to \cdots \to v_n \to T\langle \overline{a_i}^i \rangle) \in \Gamma \qquad \Gamma_{j-1} \vdash_{\text{P}} p_j : [\ \overline{a_i \mapsto \tau_i}^i\ ]v_j \rightsquigarrow \Gamma_j}{\Gamma_0 \vdash_{\text{P}} \mathsf{K}\,(p_1 \ldots p_n) : T\langle \overline{\tau_i}^i \rangle \rightsquigarrow \Gamma_n}\ \textsc{PatCon}$$

Figure 8: F²ABS typing rules – based on rules presented in [11].

σ	$::= \forall \overline{a} . \tau$	**Polymorphic Type**
Γ	$::= \epsilon \mid \Gamma, x : \tau \mid \Gamma, f : \sigma \mid \Gamma, K : \sigma$	**Environment**
θ	$::= [\overline{a \mapsto \tau}]$	**Substitution**

Figure 9: Polymorphic types, environtments, substitutions.

the environment Γ_0 the typing information of all constructors and verifies the absence of clashing on names. Its premise $\vdash_{\text{pF}} \overline{\texttt{FF}} \rightsquigarrow \Gamma_1$ can be concluded by the rule pF that parses the list $\overline{\texttt{FF}}$ of functions: it collects in the environment Γ_1 the typing information of function declarations (without checking the typing of bodies). Rule Prg concludes that a program is well-typed whenever: (i) the collected types of contructors (viz. Γ_0) are defined from type-names defined somewhere in Prg or built-in, which is formally denoted $\overline{\texttt{DD}} \propto \Gamma_0$; and (ii) the types collected in Γ_0, Γ_1 are sufficient to type the bodies (of all defined functions), namely $\Gamma_0, \Gamma_1 \vdash_{\text{FF}} \overline{\texttt{FF}}$.

Rule FF parses a function list and verifies that the body of each function is a well-typed expression in accord with the declared types. Rules Var, Fn, Con, App and Case check the types of expressions – they are the same as in [11, Fig.3]. Rules PatVar and PatCon parse nested patterns in accordance with [11]. In particular, PatCon checks (rightward) the types of all nested patterns in accord to the type of the constructor. Each check extends the environment to be used in next checks, in order to verify the compatibility of types of all involved variables.

4 DELTA-F²ABS: DOP FOR F²ABS

In Fig. 10, the syntax of Delta-F²ABS is defined: a core language formalizing DOP for F²ABS. An SPL begins with the keyword `features` followed by a list of features (i.e., Boolean variables) $\overline{\mathsf{F}}$ that allow the variability selection, and a formula Φ (after the keyword `with`) that specifies the feature model. Then, there are the base program (possible empty) and the deltas (possibly none).

`Spl`	::= `features` $\overline{\mathsf{F}}$ `with` Φ `Prg` $\overline{\texttt{Dlt}}$ `CK`	**Software Product Line**
`Dlt`	::= `delta` D $\overline{\texttt{DtO}}$ $\overline{\texttt{FnO}}$	***Delta***
`DtO`	::= `adds DD` | `removes data` T | `modifies data` T { $\overline{\texttt{KRO}}$ $\overline{\texttt{KAO}}$ }	ADT Operation
`KRO`	::= `removes` K	Constructor Remove Operation
`KAO`	::= `adds` K($\overline{v}$)	Constructor Add Operation
`FnO`	::= `adds FF` | `removes def` f | `modifies FF` | `modifiesCase` f { $\overline{\texttt{CaseO}}$ }	Function Operation
`CaseO`	::= `modifiesCase` lb { $\overline{\texttt{BRO}}$ $\overline{\texttt{BMO}}$ $\overline{\texttt{BAO}}$ $\overline{\texttt{CaseO}}$ }	Case Operation
`BRO`	::= `removes` p	Branch Remove Operation
`BMO`	::= `modifies` $p \Rightarrow e$	Branch Modify Operation
`BAO`	::= `adds` $p \Rightarrow e$	Branch Add Operation
`CK`	::= $\overline{\texttt{DAC}}$ $\overline{\texttt{DAO}}$	***Configuration Knowledge***
`DAC`	::= `delta` D `when` Φ;	Delta Activation Condition
`DAO`	::= D $\overline{\text{D}}$ < D $\overline{\text{D}}$ < D $\overline{\text{D}}$;	Delta Application Order

Figure 10: A vanilla syntax for Delta-F²ABS.

A delta comprises a set of operations on ADTs and functions. An *ADT operation* can: either add/remove an ADT definition, or modify an ADT definition by adding/removing constructors. A *function operation* can: add/remove a function definition; or modify a function definition by replacing its body with a new body (the new body may call the previous incarnation of the function via the keyword `original`); or modify some of the case-expressions occurring in the function. This latter modification is expressed by a *case-operation* which specifies: the name of the function f in which the case-expressions to be modified occur and a set of case-operations.

Each case-operation (`CaseO`) specifies: the label `lb` of the case-expression `e` to be modified; a set of remove/modify/add *branch-operations* (`BRO`/`BMO`/`BAO`) on branches of `e`; and a set of case-operations, each of which refers to a case-expression occurring either in the expression to be matched or in one of the branches of `e`. All case-expressions occurring in a function must have different labels. In order to apply a branch-operation op (∈`BRO`/`BMO`/`BAO`) on a case-expression labeled $\texttt{lb}_n$ occurring in a function `f`, the delta must specify a function operation `modifiesCase f {CaseO}` such that `CaseO` respects the label-nesting, namely

$$\texttt{CaseO} \quad \equiv \quad \texttt{modifiesCase}\, \texttt{lb}_1 \{ \cdots \texttt{modifiesCase}\, \texttt{lb}_n \{\texttt{op}\} \cdots \}$$

has to explicitly traverse the ordered full list $\texttt{lb}_1, \ldots, \texttt{lb}_{n-1}$ of the case labels in which $\texttt{lb}_n$ occurs.

Configuration knowledge `CK` provides a mapping from products to variants by describing the connection between deltas and features: it specifies an activation condition Φ (a propositional formula over features) for each delta `D` by means of a `DAC` clause; and it specifies an application ordering between deltas by means of a sequence of `DAO` clauses. Each `DAO` clause specifies a partial order over the set of deltas in terms of a total order on disjoint subsets of delta names – a `DAO` clause allows developers to express (as a partial order) dependencies between the deltas (which are usually semantic "requires" relations [2]). The overall delta application order is the union R of these partial orders – the relation R must be *consistent* (i.e., be a partial order) and *unambiguous* (i.e., all the total delta application orders that respect R generate the same variant for each product). For delta-oriented SPLs of Java-like programs, techniques for checking that R is unambiguous are available [2, 16]. These techniques can be straightforwardly adapted to Delta-F^{2}ABS SPLs.

For each set of features π, the rules for variant generation (omitted for lack of space) define a reduction $\overset{\pi}{\triangleright}$ on the SPL such that: if `Spl` $\overset{\pi}{\triangleright}$ `Prg` holds, then π is a product and `Prg` is the variant for π. The rules are stuck when an error occurs during the generation process.

5 PROPERTIES

The key property that we would like to enforce is *type safety.*

Definition 5.1 (Type-safe SPL). A Delta-F^{2}ABS SPL is *type safe* iff all its variants can be generated and are well-typed F^{2}ABS programs.

The following three properties – *label consistency*, *useless-operation absence* and *type-label uniformity* – improve the comprehensibility of an SPL and simplify type-safety checking. Label consistency guarantees that the `modifiesCases`-opeations are not ambiguous.

Definition 5.2 (Label-consistent SPL). A Delta-F^{2}ABS SPL `Spl` is *label-consistent* iff: for any two variants $\texttt{Prg}_1$ and $\texttt{Prg}_2$ of `Spl`, for any function `f` and label `l` such that `f` contains `l` in both $\texttt{Prg}_1$ and $\texttt{Prg}_2$, the path (i.e., the sequence of labels and patterns) traversed to reach `l` in `f` must be the same in both $\texttt{Prg}_1$ and $\texttt{Prg}_2$.

Example 5.3. In Figure 1, the path reaching (i.e., traversed to reach) `lbA` in the function `react` is `react.lbT.Train(pos, _, v, spec).lbA`, where "." is used to concatenate paths in the sense of Definition 5.2.

In order to define the notion of useless-operation free SPL we introduce the notion of *pure SPL.*

Definition 5.4 (Pure SPL). A Delta-F^{2}ABS SPL `Spl` is *pure* iff its base program is empty. If `Spl` is pure, then pure(`Spl`) is `Spl` itself. Otherwise, pure(`Spl`) is the SPL obtained from `Spl` by replacing its base program `Prg` with a delta with a fresh name d_{base} that consists of: (i) one add operation for each ADT declaration in `Prg`; (ii) one add operation for each function declaration in `Prg`; (iii) has application condition true; and (iv) is the first delta to be applied.

Definition 5.5 (Useless-operations and useless-operation free SPL). Let `Spl` be a Delta-F^{2}ABS SPL. A function `adds`- or `modifies`-operation, a data type constructor `adds`-operation, a branch `adds`- or `modifies`-operation of `Spl` is *useless* iff no syntax entry introduced by that operation occurs in some variant of `Spl`; and `Spl` is *useless-operation free* iff pure(`Spl`) – where each data type `adds`-operation is considered as a set of data type constructor `adds`-operations – does not contain useless operations.

Type-label uniformity ensures that each ADT contructor declaration, each function definition, and each case-expression (identified by its label) has the same type across the base program and all the deltas.

Definition 5.6 (Type-label-uniform SPL). A Delta-F^{2}ABS SPL `Spl` is *type uniform* iff:

- all declarations of the same constructor `K` declare it with the same type; and
- all declarations of the same function `f` declare it with the same type.

`Spl` is *type-label uniform* iff: it is type uniform and (for each function `f`, for each label `l`) all case-expressions labeled `l` belonging to `f` are always typed w.r.t. a same environment (same local variables with same associated types) with a same type for the matching expression and with a same type for the whole case-expression.

It worth observing that, if a Delta-F^{2}ABS SPL `Spl` is type-label uniform, then for any two variants $\texttt{Prg}_1$ and $\texttt{Prg}_2$ of `Spl`, the following statements hold:

(1) if a constructor `K` or a function `f` is declared in both $\texttt{Prg}_1$ and $\texttt{Prg}_2$, it must be declared with the same type in both variants; and
(2) if both $\texttt{Prg}_1$ and $\texttt{Prg}_2$ declare a function `f` containing a case-expression with the same label, then these two case-expressions have the same type and match-expressions of the same type.

The vice-versa (i.e., property 1 and 2 imply label-type-uniformity) holds whenever `Spl` is useless-operation free.

The SPL illustrated in Sect. 2 is type-safe, label-consistent, useless-operation free, and type-label uniform.

6 FAMILY-BASED CHECKINGS

We formalize the notion of paths (already used in Definition 5.2) to identify syntactic elements E occurring in an SPL `Spl` together with the "locality" where E is expected to (possibly) occur in variants.

Definition 6.1 (Paths in an SPL). Let $\mathcal{A}$ be the alphabet where letters are type names T, constructors `K`, function names `f`, case labels l and patterns p. A *path* ρ is a (possibly empty) sequence of letters in $\mathcal{A}$ that falls in one of the following categories: (i) The name of a datatype T is the path to T (because datatype definitions are outer localities); (ii) $T.$`K` is the path to the definition of the constructor `K` in the datatype T; (iii) The name of a function `f`

is the path to `f` (because function definitions are outer localities); (iv) A case-expression labeled `lb` in a function `f` is identified by a path of the form $\texttt{f}.((1^+.(p|\epsilon))^*).\texttt{lb}$ (where ϵ is the empty path) that locates (nesting-respecting) all and only case-expressions (using their labels) and branches (using their patterns) in which the `lb`-labeled case-expression is nested in; and (v) A branch $p \Rightarrow e$ of a case-expression labeled `lb` in a function `f` is identified by the path $\texttt{f}.\rho'.\texttt{lb}.p$, for some ρ'.

Given a path ρ, we denote by *prefix*(ρ) the set of prefixes of ρ. We denote $\mathsf{P}(\texttt{Spl})$ the set of all paths occurring in `Spl` (i.e., the paths identifying the relevant syntactic elements in `Spl`).

For any path ρ, the path obtained from ρ by removing all patterns is by denoted *rmp*(ρ). It is inductively defined as follows:

$$rmp(\varepsilon) = \varepsilon \quad rmp(\rho.a) = \begin{cases} rmp(\rho) & \text{if } a \text{ is a pattern} \\ rmp(\rho).a & \text{otherwise.} \end{cases}$$

Some of our analysis traverse the input SPL to construct tables Θ (i.e., mappings from keys to values) collecting information that are exploited by subsequent analyses.

We define an operation to extend a table Θ with a pair $k \mapsto v$. The operation fails if k already occurs in the domain of Θ and $\Theta(k)$ is different from v.

$$\text{add}_\text{G}(\Theta, k, v) = \begin{cases} \Theta \cup \{k \mapsto v\} & \text{if } k \notin \text{dom}(\Theta) \\ \Theta & \text{if } v = \Theta(k). \end{cases}$$

6.1 Non-Variable Checkings

6.1.1 Label Consistency. This analysis checks label-consistency (Definition 5.2) and that each label `lb` occurs at most once in each function. W.l.o.g., we assume that in the input SPL the deltas $\overline{\texttt{Dlt}}$ are listed in an order compatible with the one specified by configuration knowledge `CK`. Figure 11 presents the rules of this analysis, where tables Θ map $(\texttt{f}, \texttt{l})$ (viz. pairs function-name and label) to paths ρ. They have three possible judgemens: the judgement $\vdash_{\text{lc}} \texttt{Spl} : \Theta$ states that the SPL `Spl` has a valid label forest, represented by the table Θ; the judgement $\Theta \vdash_{\text{lc}} E : \Theta'$ states that the labels in E are not conflicting with Θ and that extending Θ with the (possible) additional labels in E yield Θ'; and the judgemens $\Theta, \rho \vdash_{\text{lc}} E : \Theta'$ states the consistency of E, assumed to be nested in ρ.

Most rules in Figure 11 simply traverse the input, looking for cases and their labels. Rule L:7 checks the consistency of the encountered label `l` with paths in Θ and then proceeds to check all its sub-expressions in their nesting paths. Rule L:13 is driven by a path $\texttt{f}.\rho$ that does not contain the nesting patterns (it is just the sequence of labels of the `modifiesCase` in the delta under analysis) – the sequence of patterns (identifying the nesting branches) is recovered by using Θ. Rule L:13 looks for consistent/new paths in the encountered case-operations $\overline{\texttt{BMO}}$ (by expoiting rule L:14), $\overline{\texttt{BAO}}$ (by exploiting rule L:15) and $\overline{\texttt{CaseO}}$ (by exploiting rule L:13).

Theorem 6.2 (Label Consistency). *Consider the following two statements over an SPL* `Spl`*:*

1. *The judgment* $\vdash_{\text{lc}} \texttt{Spl} : \Theta$ *(for some* Θ*) holds.*
2. `Spl` *is label-consistent and each label occurs at most once in each function.*

Then statement 1 implies statement 2. Moreover, if `Spl` *is useless-operation free, then statement 2 implies statetement 1.*

6.1.2 Type Uniformity. The type uniformity of an SPL `Spl` can be straightfowardly checked by collecting in a table Θ all the types of all constructors and functions (declared in the base programs and the deltas) and verifying that Θ maps each entry to a single polymorphic type. If `Spl` is type uniform we denote $\Gamma_{\texttt{Spl}}$ the table Θ, which is indeed the environment associating to functions and constructors their unique polymorphic types in `Spl`.

In the rest of this document, we assume type uniformity for each considered SPL.

6.1.3 Type-Label Uniformity and Partial Typing. Partial typing adapts (neglecting dataypes) the type system of Fig. 8 to check that an SPL `Spl` is type-label uniform and that all the functios and constructors are used according the their type in $\Gamma_{\texttt{Spl}}$.

Before introducing the rules for partial typing, we recall that in a case-expression $\texttt{case}\ e\ \texttt{as}\ l\ \{\ \overline{p_i \Rightarrow e_i}\ \}$: (i) the "input" expression e (with a type ν) is matched against all patterns p_i; (ii) the whole expression has an "output" type τ which is the type of all expressions e_i in the branches $\overline{p_i \Rightarrow e_i}$ (c.f. Fig. 8); (iii) each pattern p_i binds variables in the branch $p_i \Rightarrow e_i$, the actual values of these variables is obtained by the match against the "input" expression e, and the typing rules $\Gamma \vdash_{\text{P}} p : \nu \rightsquigarrow \Gamma'$ of Fig. 8 extend Γ with types for bound variables. Therefore, in many respects, a case-expression looks like a function and we can associate to it the type $\nu \to \tau$. For each `lb`-labeled case-expression in a function `f`, we use type tables Θ to store the typing of available local variables (parameters of `f` and outer patterns) together with the type $\nu \to \tau$. In this way, local environments becomes available to checks that delta declarations modifying case are uniform and well typed as required by type-label uniformity (Definition 5.6).

Figure 12 presents the rules for partial typing. Many of them straightforwardly adapt the rules of the type system in Fig. 8. In particular: rule FF corresponds to rule U:2; and rules Var, Con, Fn, App, Case correspond to rules U:3, U:4, U:5, U:6, U:7, respectively.

The rules for partial typing have four possible judgments: (1) the judgment in the conclusion of U:1 is about a whole SPL; (2) judgments of the form $\Theta \vdash_{\text{u}} E : \Theta'$ state that E has no case-declaration conflicting with Θ and that Θ' is the extension of Θ with the case-declarations in E; (3) judgments of the form $\Theta, \rho \vdash_{\text{u}} E : \Theta'$ state that E lives in the branch ρ, that E has no case-declaration conflicting with Θ, and that Θ' is the extension of Θ with the case-declarations in E; and (4) judgments of the form $\Theta, \Gamma, \rho \vdash_{\text{u}} e : \nu, \Theta'$ state that E lives in the branch ρ, and E has no case-declaration conflicting with Θ when local variables are uniformly typed by Γ and that Θ' is the extension of Θ with the case-declarations in E.

W.l.o.g., we assume that in the input SPL the deltas $\overline{\texttt{Dlt}}$ are listed in an order compatible with the one specified by configuration knowledge `CK`. Rule U:2 stores function parameter as the local variables in an environment, and record the function name as starting path. Rules U:3 to U:6 simply traverse expressions and check that they are well typed w.r.t. the type table and typing environment Γ. Rule U:7 manages the case expression by traversing its sub-expressions and add (or check the consistency of) its type and its local environment in the type table. The rest of the rules deals with deltas and delta operations. In particular, rules U:9 and U:11 forward the function declarations to the rule U:2. Rules U:12 predispose the initial path where local variables can be found in Θ.

$$\frac{\Theta_1 = \emptyset \quad \overline{\Theta_i \vdash_{\mathrm{lc}} \texttt{FF}_i : \Theta_{i+1}} \quad \overline{\Theta_i \vdash_{\mathrm{lc}} \texttt{Dlt}_i : \Theta_{i+1}}}{\vdash_{\mathrm{lc}} \texttt{features}\ \overline{\mathsf{F}}\ \texttt{with}\ \Phi\ \overline{\texttt{DD}}\ \texttt{FF}_1 \ldots \texttt{FF}_{n'}\ \texttt{Dlt}_{n'+1} \ldots \texttt{Dlt}_{n''}\ \texttt{CK} : \Theta_{n''+1}}\ \text{L:1} \qquad \frac{\Theta, \mathsf{f} \vdash_{\mathrm{lc}} e : \Theta'}{\Theta \vdash_{\mathrm{lc}} \texttt{def}\ \tau\ \mathsf{f}\langle a_1, \ldots a_n\rangle(\mathsf{x}_1 : v_1, \ldots, \mathsf{x}_m : v_m) = e : \Theta'}\ \text{L:2}$$

$$\frac{}{\Theta, \rho \vdash_{\mathrm{lc}} \mathsf{x} : \Theta}\ \text{L:3} \qquad \frac{\overline{\Theta_i, \rho \vdash_{\mathrm{lc}} e_i : \Theta_{i+1}}}{\Theta_1, \rho \vdash_{\mathrm{lc}} \mathsf{K}(e_1, \ldots, e_n) : \Theta_{n+1}}\ \text{L:4} \qquad \frac{}{\Theta, \rho \vdash_{\mathrm{lc}} \mathsf{f} : \Theta}\ \text{L:5} \qquad \frac{\Theta_1, \rho \vdash_{\mathrm{lc}} e_1 : \Theta_2 \quad \Theta_2, \rho \vdash_{\mathrm{lc}} e_2 : \Theta_3}{\Theta_1, \rho \vdash_{\mathrm{lc}} e_1\ e_2 : \Theta_3}\ \text{L:6}$$

$$\frac{\rho' = \mathsf{f}.\rho.\mathsf{l} \quad \Theta_0 = \mathrm{add}_{\mathrm{G}}(\Theta, (\mathsf{f}, \mathsf{l}), \rho') \quad \Theta_0, \rho' \vdash_{\mathrm{lc}} e : \Theta_1 \quad \overline{\Theta_i, \rho'.p_i \vdash_{\mathrm{lc}} e_i : \Theta_{i+1}}}{\Theta, \mathsf{f}.\rho \vdash_{\mathrm{lc}} \texttt{case}\ e\ \texttt{as}\ \mathsf{l}\ \{\ p_1 \Rightarrow e_1\ \ldots\ p_n \Rightarrow e_n\ \} : \Theta_{n+1}}\ \text{L:7} \qquad \frac{\overline{\Theta_i \vdash_{\mathrm{lc}} \texttt{FnO}_i : \Theta_{i+1}}}{\Theta_1 \vdash_{\mathrm{lc}} \texttt{delta}\ \mathsf{D}\ \overline{\texttt{DtO}}\ \texttt{FnO}_1 \ldots \texttt{FnO}_m : \Theta_{m+1}}\ \text{L:8} \qquad \frac{\Theta \vdash_{\mathrm{lc}} \texttt{FF} : \Theta'}{\Theta \vdash_{\mathrm{lc}} \texttt{adds}\ \texttt{FF} : \Theta'}\ \text{L:9}$$

$$\frac{}{\Theta \vdash_{\mathrm{lc}} \texttt{removes}\ \texttt{def}\ f : \Theta}\ \text{L:10} \qquad \frac{\Theta \vdash_{\mathrm{lc}} \texttt{FF} : \Theta'}{\Theta \vdash_{\mathrm{lc}} \texttt{modifies}\ \texttt{FF} : \Theta'}\ \text{L:11} \qquad \frac{\overline{\Theta_i, \mathsf{f} \vdash_{\mathrm{lc}} \texttt{CaseO}_i : \Theta_{i+1}}}{\Theta_1 \vdash_{\mathrm{lc}} \texttt{modifiesCase}\ \mathsf{f}\ \{\ \texttt{CaseO}_1 \ldots \texttt{CaseO}_n \} : \Theta_{n+1}}\ \text{L:12} \qquad \frac{\Theta, \rho.p \vdash_{\mathrm{lc}} e : \Theta'}{\Theta, \rho \vdash_{\mathrm{lc}} \texttt{modifies}\ p \Rightarrow e : \Theta'}\ \text{L:14}$$

$$\frac{\rho_1 = \mathsf{f}.\rho.\mathsf{l} \quad \rho_2 = \Theta(\mathsf{f}, \mathsf{l}) \quad rmp(\rho_2) = \rho_1 \quad \overline{\Theta_i, \rho_2 \vdash_{\mathrm{lc}} \texttt{BMO}_i : \Theta_{i+1}} \quad \overline{\Theta_i, \rho_2 \vdash_{\mathrm{lc}} \texttt{BAO}_i : \Theta_{i+1}} \quad \overline{\Theta_i, \rho_1 \vdash_{\mathrm{lc}} \texttt{CaseO}_i : \Theta_{i+1}}}{\Theta, \mathsf{f}.\rho \vdash_{\mathrm{lc}} \texttt{modifiesCase}\ \mathsf{l}\ \{\ \overline{\texttt{BRO}}\ \texttt{BMO}_1 \ldots \texttt{BMO}_m \texttt{BAO}_{m+1} \ldots \texttt{BAO}_n\ \texttt{CaseO}_{n+1} \ldots \texttt{CaseO}_k\ \} : \Theta_{k+1}}\ \text{L:13} \qquad \frac{\Theta, \rho.p \vdash_{\mathrm{lc}} e : \Theta'}{\Theta, \rho \vdash_{\mathrm{lc}} \texttt{adds}\ p \Rightarrow e : \Theta'}\ \text{L:15}$$

Figure 11: Label Consistency

$$\frac{\Theta_1 = \emptyset \quad \overline{\Theta_i \vdash_{\mathrm{u}} \texttt{FF}_i : \Theta_{i+1}} \quad \overline{\Theta_i \vdash_{\mathrm{u}} \texttt{Dlt}_i : \Theta_{i+1}}}{\vdash_{\mathrm{u}} \texttt{features}\ \overline{\mathsf{F}}\ \texttt{with}\ \Phi\ \overline{\texttt{DD}}\ \texttt{FF}_1 \ldots \texttt{FF}_n\ \texttt{Dlt}_{n+1} \ldots \texttt{Dlt}_{n'}\ \texttt{CK} : \Theta_{n'+1}}\ \text{U:1} \qquad \frac{\Theta_1, [\overline{\mathsf{x}_i \mapsto v_i}], \mathsf{f} \vdash_{\mathrm{u}} e : \tau, \Theta_2}{\Theta_1 \vdash_{\mathrm{u}} \texttt{def}\ \tau\ \mathsf{f}\langle a_1, \ldots a_n\rangle(\mathsf{x}_1 : v_1, \ldots, \mathsf{x}_m : v_m) = e : \Theta_2}\ \text{U:2} \qquad \frac{\Gamma(\mathsf{x}) = v}{\Theta, \Gamma, \mathsf{f}.\rho \vdash_{\mathrm{u}} \mathsf{x} : v, \Theta}\ \text{U:3}$$

$$\frac{\mathsf{s} = [\overline{a_i \mapsto \tau_i}] \quad \overline{\Theta_i, \Gamma, \mathsf{f}.\rho \vdash_{\mathrm{u}} e_i : \sigma(v_i), \Theta_{i+1}} \quad \Gamma_{\texttt{Spl}}(\mathsf{K}) = \forall \overline{a_i}^i . v_1 \to \cdots \to v_n \to T\langle \overline{a_i}^i \rangle}{\Theta_1, \Gamma, \mathsf{f}.\rho \vdash_{\mathrm{u}} \mathsf{K}(e_1, \ldots, e_n) : T\langle \tau_1, \ldots, \tau_m \rangle, \Theta_{n+1}}\ \text{U:4} \qquad \frac{\mathsf{s} = [\overline{a_i \mapsto \tau_i}] \quad \Gamma_{\texttt{Spl}}(\mathsf{f}) = \forall \overline{a_i}^i . \tau}{\Theta, \Gamma, \mathsf{f}.\rho \vdash_{\mathrm{u}} \mathsf{f} : \mathsf{s}(\tau), \Theta}\ \text{U:5}$$

$$\frac{\Theta_1, \Gamma, \mathsf{f}.\rho \vdash_{\mathrm{u}} e_1 : \tau_1 \to \tau_2, \Theta_2 \quad \Theta_2, \Gamma, \mathsf{f}.\rho \vdash_{\mathrm{u}} e_2 : \tau_1, \Theta_3}{\Theta_1, \Gamma, \mathsf{f}.\rho \vdash_{\mathrm{u}} e_1\ e_2 : \tau_2, \Theta_3}\ \text{U:6} \qquad \frac{\begin{array}{c}\Theta_0, \Gamma, \mathsf{f}.\rho.\mathsf{l} \vdash_{\mathrm{u}} e : v, \Theta_1 \quad \overline{\Gamma \vdash_{\mathrm{P}} p_i : v \rightsquigarrow \Gamma_i} \quad \overline{\Theta_i, \Gamma_i, \mathsf{f}.\rho.\mathsf{l}.p_i \vdash_{\mathrm{u}} e_i : \tau, \Theta_{i+1}} \\ \Theta_r = \mathrm{add}_{\mathrm{G}}(\Theta_{n+1}, (\mathsf{f}, \mathsf{l}), (\Gamma, v \to \tau))\end{array}}{\Theta_0, \Gamma, \mathsf{f}.\rho \vdash_{\mathrm{u}} \texttt{case}\ e\ \texttt{as}\ \mathsf{l}\ \{\ p_1 \Rightarrow e_1\ \ldots\ p_n \Rightarrow e_n\ \} : \tau, \Theta_r}\ \text{U:7}$$

$$\frac{\overline{\Theta_i \vdash_{\mathrm{u}} \texttt{FnO}_i : \Theta_{i+1}}}{\Theta_1 \vdash_{\mathrm{u}} \texttt{delta}\ \mathsf{D}\ \overline{\texttt{DtO}}\ \texttt{FnO}_1 \ldots \texttt{FnO}_n : \Theta_{n+1}}\ \text{U:8} \qquad \frac{\Theta \vdash_{\mathrm{u}} \texttt{FF} : \Theta'}{\Theta \vdash_{\mathrm{u}} \texttt{adds}\ \texttt{FF} : \Theta'}\ \text{U:9} \qquad \frac{}{\Theta \vdash_{\mathrm{u}} \texttt{removes}\ \texttt{def}\ f : \Theta}\ \text{U:10} \qquad \frac{\Theta \vdash_{\mathrm{u}} \texttt{FF} : \Theta'}{\Theta \vdash_{\mathrm{u}} \texttt{modifies}\ \texttt{FF} : \Theta'}\ \text{U:11}$$

$$\frac{\overline{\Theta_i, \mathsf{f} \vdash_{\mathrm{u}} \texttt{CaseO}_i : \Theta_{i+1}}}{\Theta_1 \vdash_{\mathrm{u}} \texttt{modifiesCase}\ \mathsf{f}\ \{\ \texttt{CaseO}_1 \ldots \texttt{CaseO}_n \} : \Theta_{n+1}}\ \text{U:12} \qquad \frac{\overline{\Theta_i, \rho.\mathsf{l} \vdash_{\mathrm{u}} \texttt{BMO}_i : \Theta_{i+1}} \quad \overline{\Theta_i, \rho.\mathsf{l} \vdash_{\mathrm{u}} \texttt{BAO}_i : \Theta_{i+1}} \quad \overline{\Theta_i, \rho.\mathsf{l} \vdash_{\mathrm{u}} \texttt{CaseO}_i : \Theta_{i+1}}}{\Theta_1, \rho \vdash_{\mathrm{u}} \texttt{modifiesCase}\ \mathsf{l}\ \{\ \overline{\texttt{BRO}}\ \texttt{BMO}_1 \ldots \texttt{BMO}_m\ \texttt{BAO}_{m+1} \ldots \texttt{BAO}_n\ \texttt{CaseO}_{n+1} \ldots \texttt{CaseO}_k\ \} : \Theta_{k+1}}\ \text{U:13}$$

$$\frac{\Theta(\mathsf{f}, \mathsf{l}) = (\Gamma, v \to \tau) \quad \Gamma \vdash_{\mathrm{P}} p : v \rightsquigarrow \Gamma' \quad \Theta, \Gamma', \mathsf{f}.\rho.\mathsf{l} \vdash_{\mathrm{u}} e : \tau, \Theta'}{\Theta, \mathsf{f}.\rho.\mathsf{l} \vdash_{\mathrm{u}} \texttt{modifies}\ p \Rightarrow e : \Theta'}\ \text{U:14} \qquad \frac{\Theta(\mathsf{f}, \mathsf{l}) = (\Gamma, v \to \tau) \quad \Gamma \vdash_{\mathrm{P}} p : v \rightsquigarrow \Gamma' \quad \Theta, \Gamma', \mathsf{f}.\rho.\mathsf{l} \vdash_{\mathrm{u}} e : \tau, \Theta'}{\Theta, \mathsf{f}.\rho.\mathsf{l} \vdash_{\mathrm{u}} \texttt{adds}\ p \Rightarrow e : \Theta'}\ \text{U:15}$$

Figure 12: Partial Typing

Rules U:14 and U:15 predispose the local variables for possible recursive call to the rule U:7 in presence of case-expressions. Note that the branch removal operations are simply skipped, since they do not declare anything and do not contain any expressions that need to be type-checked.

THEOREM 6.3 (PARTIAL TYPING). *Let* `Spl` *be a label-consistent type-uniform SPL. Consider the following three statements over* `Spl`*:*

(1) The judgment $\vdash_{\mathrm{u}}$ `Spl` $: \Theta$ *(for some* Θ*) holds.*
(2) `Spl` *is label-type uniform.*
(3) Each variant $\texttt{Prg}_i = \overline{\texttt{DD}}\,\overline{\texttt{FF}}$ *of* `Spl` *is such that* $\Gamma_{\texttt{Spl}} \vdash_{FF} \overline{\texttt{FF}}$ *(c.f. Fig. 8).*

Then statement 1 implies statements 2 and 3. Moreover, if `Spl` *has no useless operations, then statement 2 and 3 imply statement 3.*

6.2 Variable Checkings

6.2.1 Getters. The family-based analyses presented in this section use some auxiliary functions (aka getters) that extracts information from SPLs.

Definition 6.4 (Getters on FM and CK). Given an SPL `Spl`, we denote by $fm(\texttt{Spl})$ the propositional formula (over feature names) specifying the feature model of `Spl`. Moreover, we denote by $act(\texttt{Spl})$ the propositional formula (over feature names and delta names) formed by the conjunction of bi-implications equating each delta name to its activation conditions.

For the sake of simplicity, we assume that in what follows the considered SPL `Spl` is pure and label-consistent (c.f. Section 6.1.1), so that Θ is the table that stores all path identifying the labels of `Spl`, namely $\vdash_{\mathrm{lc}}$ `Spl` $: \Theta$. Moreover, we denote $<$ a total order on deltas compatible with the configuration knowledge of `Spl`.

We use paths, as defined in Definition 6.1, to identify the syntactic elements E considered in declarations of `Spl` together with the "locality" where E is expected to (possibly) occur in variants. Notably, the prefixes of a path ρ allows us to identify the super-localities whose modification may affect delta-operations on syntactic elements identified by ρ.

In accord with Definition 6.1, we have the following kinds of paths: a datatype is identified by its name; a constructor `K` of a datatype T is identified by T.`K`; a function is identified by its name; a case labeled `lb` in a function `f` is identified by $\Theta($`f`$,$ `lb`$) = $ `f`$.\rho.$`lb` (such that $\vdash_{\text{lc}}$ `Spl` $: \Theta$ and ρ is described by the regular expression $(1^+.(p|\epsilon))^*$); and, a branch $p \Rightarrow e$ in case labeled `lb` of a function `f` is identified by $\rho.p$ such that $\Theta($`f`$,$ `lb`$) = \rho$.

We define the following getters on paths.

Definition 6.5 (Getters on paths). Given an SPL `Spl` and a path $\rho \in \mathsf{P}($`Spl`$)$, we denote by:

- $add($`Spl`$, \rho)$ the set of delta names d that add the path ρ in `Spl`;
- $rem($`Spl`$, \rho)$ the set of delta names d that remove the path ρ in `Spl`; and
- $mod($`Spl`$, \rho)$ the set of delta names d that modify the path ρ in `Spl`.

It is worth observing that the name of the variability operations in a delta `d` makes evident the getters that return `d`. For instance, for ADT-operations: (i) if `d` contains **adds DD** (adding a datatype named T) then `d` $\in add($`Spl`$, T)$; (ii) if `d` contains **removes data**T then `d` $\in rem($`Spl`$, T)$; (iii) if `d` contains **modifies data** T{ **removes** `K_0` **adds** `K_1`$(\overline{V})$} then `d` $\in mod($`Spl`$, T)$, `d` $\in rem($`Spl`$, T.K_0)$ and `d` $\in add($`Spl`$, T.K_1)$. Moreover, the getters in Definition 6.5 behaves on a delta `d` containing **adds FF**, **removes def** `f`, **modifies FF**, **BRO**, **BMO**, **BAO** as follows: (i) if `d` contains **adds FF** then, for all ρ occurring in **FF** (included $\rho =$ `f`), `d` $\in add($`Spl`$, \rho)$; (ii) if `d` contains `modify` **FF** (acting on the function `f`), then `d` $\in rem($`Spl`$,$ `f`$)$ and, for all ρ occuring in **FF** (included $\rho =$ `f`), `d` $\in add($`Spl`$, \rho)$; (iii) if `d` contains **BAO** of the form **adds** $p \Rightarrow e$ (acting on the function `f`, so it is included in a **modifiesCase** `f`) aiming to add the branch to the case `l` (so its nearest **modifiesCase** is labeled `l`) and $\Theta($`f`$,$ `l`$) = \rho'$, then, for all ρ occurring in $p \Rightarrow e$ (included $\rho = p$), `d` $\in add($`Spl`$, \rho'.\rho)$; (iv) if `d` contains **BMO** of the form **modifies** $p \Rightarrow e$ (acting on the function `f`, so it is included in a **modifiesCase** `f`) aiming to modify the branch in the case `l` (so its nearest **modifiesCase** is labeled `l`) and $\Theta($`f`$,$ `l`$) = \rho'$, then `d` $\in rem($`Spl`$, \rho'.p)$ and, for all ρ occurring in $p \Rightarrow e$ (included $\rho = p$), `d` $\in add($`Spl`$, \rho.\rho')$; (v) if `d` contains either **removes def** `f` or **BRO** (acting on the path ρ), then $d \in rem($`Spl`$, \rho)$. It is worth to note, that if `d` contains **modifiesCase** on `l` nested in a **modifiesCase** `f` and $\Theta($`f`$,$ `l`$) = \rho$, then `d` $\in mod($`Spl`$, \rho')$ for all $\rho' \in prefix(\rho)$.

6.2.2 Applicability consistency.

Addition Operation. Given an SPL `Spl`, the constraint for checking that no addition operation of a path $\rho \in \mathsf{P}($`Spl`$)$ fails is as follows:

$$predA(\texttt{Spl}, \rho) = \bigwedge_{d \neq d'} \Big(d \wedge d' \Rightarrow \bigvee_{d''} d''\Big)$$
$$\text{with } \begin{cases} d, d' \in add(\texttt{Spl}, \rho), d'' \in \bigcup_{\rho' \in prefix(\rho)} rem(\texttt{Spl}, \rho') \\ \text{and } d' < d'' \leq d \end{cases}$$

This constraint states that if two deltas add the same path, then there must be a third one in between that removes it (possibly $d'' = d$).

Removal Operation. Given an SPL `Spl`, the constraint for checking that no removal operation of a path $\rho \in \mathsf{P}($`Spl`$)$ fails is as follows:

$$predR(\texttt{Spl}, \rho) = \bigwedge_{d} \Bigg(d \Rightarrow \Big(\bigvee_{d''} \big(d'' \wedge \bigwedge_{d'} \neg d'\big)\Big)\Bigg)$$
$$\text{with } \begin{cases} d \in rem(\texttt{Spl}, \rho), d' \in \bigcup_{\rho' \in prefix(\rho)} \big(rem(\texttt{Spl}, \rho') - add(\texttt{Spl}, \rho')\big), \\ d'' \in add(\texttt{Spl}, \rho) \text{ and } d'' < d' < d \end{cases}$$

This constraint states that for a removal operation to succeed (in delta d), there must be a previous delta d'' that added the path to remove, with no other delta d' in between removing it first.

Modification Operation. Given an SPL `Spl`, the constraint for checking that no modification operation of a path $\rho \in \mathsf{P}($`Spl`$)$ fails is as follows:

$$predM(\texttt{Spl}, \rho) = \bigwedge_{d} \Bigg(d \Rightarrow \Big(\bigvee_{d''} \big(d'' \wedge \bigwedge_{d'} \neg d'\big)\Big)\Bigg)$$
$$\text{with } \begin{cases} d \in mod(\texttt{Spl}, \rho), d' \in \bigcup_{\rho' \in prefix(\rho)} \big(rem(\texttt{Spl}, \rho') - add(\texttt{Spl}, \rho')\big), \\ d'' \in add(\texttt{Spl}, \rho) \text{ and } d'' < d' < d \end{cases}$$

This constraint has the same structure as $predR($`Spl`$, \rho)$ before: for a modification operation to succeed (in delta d), there must be a previous delta d'' that added the path to remove, with no other delta d' in between removing it first.

Full Applicability Constraint. We can now combine all the previous constraints to ensure that all delta operations are valid:

$$predAPP(\texttt{Spl}) = \bigwedge_{\rho \in \mathsf{P}(\texttt{Spl})} \Big(predA(\texttt{Spl}, \rho) \wedge predR(\texttt{Spl}, \rho) \wedge predM(\texttt{Spl}, \rho\Big)$$

Finally, we can bind this constraint to the variability model of the SPL to obtain the formula

$$(fm(\texttt{Spl}) \wedge act(\texttt{Spl})) \Rightarrow predAPP(\texttt{Spl})$$

This formula states that given a product π (i.e., a model of $fm($`Spl`$)$), and extend it to the set of delta's activated by π (i.e., a model of $act($`Spl`$)$), then if the resulting model validates the constraints, then all delta operations triggered by the product π will succeed, i.e., the corresponding variant can be generated. This property is formalized in the following theorem.

Theorem 6.6 (Applicability consistency). *The following two statements on an SPL* `Spl` *are equivalent:*

1. *The constraint $(fm(\texttt{Spl}) \wedge act(\texttt{Spl})) \Rightarrow predAPP(\texttt{Spl})$ is valid.*
2. *All variants of* `Spl` *can be generated.*

6.2.3 Pattern Compatibility.

This analysis checks that during the generation of a variant, no incompatible (i.e., violating the restrictions R1 and R2 given in Sect. 1) patterns can be used in the same case-expression.

Definition 6.7 (Pattern incompatibility). Two patterns p_1 and p_2 are *incompatible* (written $\downdownarrows (p_1, p_2)$) iff either:

- p_1 and p_2 are not variables and there exists σ_1 and σ_2 such that $\sigma_1(p_1) = \sigma_2(p_2)$; or
- p_1 and p_2 are variables.

Given an SPL $\texttt{Spl}$, we define the getter on SPL such that

$$pat(\texttt{Spl}, \texttt{f}, l) = \{p \mid \exists \rho,\ \texttt{f}.\rho.l.p \in \mathsf{P}(\texttt{Spl})\}.$$

Given an SPL $\texttt{Spl}$, the fact that a path $\rho \in \mathsf{P}(\texttt{Spl})$ is present in a variant is specified by the following constraint:

$$Pre(\texttt{Spl}, \rho) = \bigvee_{d} \Big(d \wedge \big(\bigwedge_{d'} \neg d\big)\Big) \text{ with } \begin{cases} d \in add(\texttt{Spl}, \rho), \\ d' \in \bigcup_{\rho' \in prefix(\rho)} rem(\texttt{Spl}, \rho') \\ \text{and } d < d' \end{cases}$$

The following constraint specifies that a variant must not contain any incompatible pattern, simply by stating that two incompatible patterns cannot be present together.

$$predC(\texttt{Spl}) = \bigwedge_{\texttt{f}.\rho.l \in \mathsf{P}(\texttt{Spl})} \bigwedge_{\substack{p_1 \neq p_2 \in pat(\texttt{Spl}, \texttt{f}, l) \\ \downdownarrows(p_1, p_2)}} \neg \begin{pmatrix} Pre(\texttt{Spl}, \texttt{f}.\rho.l.p_1) \\ \Leftrightarrow Pre(\texttt{Spl}, \texttt{f}.\rho.l.p_2) \end{pmatrix}$$

Theorem 6.8 (Pattern compatibility). *Consider the following two statements over and SPL* $\texttt{Spl}$*:*

(1) The constraint $(fm(\texttt{Spl}) \wedge act(\texttt{Spl})) \Rightarrow predC(\texttt{Spl})$ *is valid.*
(2) No generable variant of $\texttt{Spl}$ *has incompatible patterns in a case-expression.*

Then statement 1 implies statement 2. Moreover, if all variants of $\texttt{Spl}$ *can be generated, then statement 2 implies statement 1.*

6.2.4 Dependency Analysis. The dependency analysis is formalized by the judgment $\vdash \texttt{Spl} : \Phi$, defined by the rules in Fig. 13, deriving a constraint Φ stating that all elements (types, constructors and function) used in a variant of the SPL $\texttt{Spl}$ are declared in that variant.

We reuse the predicate $Pre(\texttt{Spl}, \rho)$ defined in Section 6.2.3 in order to check that ρ is added by some deltas and not subsequently removed. But we need another predicate $Pre(\texttt{Spl}, \rho, d)$ asserting that if d is activated then there exists another delta d' which is activated and never removed taking into account `original`.

Given an SPL $\texttt{Spl}$, a delta d adding a path $\rho \in \mathsf{P}(\texttt{Spl})$, the constraint stating that the version of ρ given by d is present in a variant is as follows:

$$Pre(\texttt{Spl}, \rho, d) = d \wedge \left(\bigwedge_{d'} d' \Rightarrow \bigvee_{\rho'', d''} Pre(\texttt{Spl}, \rho'', d'')\right)$$

$$\text{with: } \begin{cases} d' \in \bigcup_{\rho' \in prefix(\rho)} rem(\texttt{Spl}, \rho') \\ \rho'', d'' \in kept(\texttt{Spl}, \rho) \text{ and } d < d'' \leq d' \end{cases}$$

where $kept(\texttt{Spl}, \rho)$ is the set of pairs (ρ'', d'') such that:

$$\begin{cases} \text{there is } \texttt{f} \text{ s.t. } \texttt{f} \in prefix(\rho) \cap prefix(\rho'') \\ \text{and } d'' \in add(\rho'') \\ \text{and } \rho'' \text{ is a maximal path in } d'' \text{ where an } \texttt{original} \text{ occurs} \end{cases}$$

This formula states that for the declaration of ρ given in d to be accessible in a variant, we need that d must be activated, and that for any delta d' that remove ρ after d, there must be another delta d'' in between that keeps ρ visible through a call to `original`. More precisely, d'' must introduce a path ρ'' that: (i) identifies a call to `original`, i.e., removing it is equivalent to removing the `original` call (this is encoded with ρ'' being a *maximal path* containing the `original` call); (ii) is located in the new body of the same function `f` where ρ was located; and (iii) is accessible in the variant (i.e., $Pre(\texttt{Spl}, \rho'', d'')$ must hold).

The rules in Figure 13 have three possible judgments: a judgment of the form $\vdash E : \Phi$ states that the syntactic entry E generates the constraint Φ; a judgment of the form $d \vdash E : \Phi$ states that the syntactic entry E in the delta d generates the constraint Φ; and statements of the form $d, \rho \vdash_{\text{u}} E : \Phi$ states that the syntactic entry E living in the path ρ generates the constraint Φ.

Most of rules in Figure 13 simply accumulates constraints from inner syntax node, or return **true** because that syntax entry contains no dependency. There are only three rules that actually introduce dependency in the generated constraint Φ: rule D:6 states that when a datatype T is used, then it must exists; rule D:8 states that when a constructor κ is used, then it must exists; and rule D:9 states that when a function `f` is used, then it must exists.

6.2.5 Type-Safety Analysis. The type-safety analysis relies on the analyses presented so far. It is illustrated by the following theorem.

Theorem 6.9 (Type-safety). *Consider a type-uniform SPL* $\texttt{Spl}$*, a type table* Θ *and a constraint* Φ *such that both* $\Theta' \vdash_{\text{u}} \texttt{Spl} : \Theta$ *and* $\vdash \texttt{Spl} : \Phi$ *hold; and consider the following two statements over* $\texttt{Spl}$*:*

(1) The constraint $(fm(\texttt{Spl}) \wedge act(\texttt{Spl})) \Rightarrow \Phi$ *is valid.*
(2) All generable variants of $\texttt{Spl}$ *are well typed.*

Then statement 1 implies statement 2. Moreover, if all variant of $\texttt{Spl}$ *can be generated, then statement 2 implies statement 1.*

7 INTEGRATION WITH CLASS-BASED DOP

ABS (https://abs-models.org/) is multi paradigm modelling language which supports DOP for its class-based object-oriented sublanguage. Its toolchain implements type-uniformity, pre-typing and applicability-consistency family-based checks, while dependency-consistency is currently under development [7]. In this section we briefly outline how delta-oriented constructs for functional programs and the associated family-based checks proposed in this paper can be smoothly integrated with their counterparts for class-based object-oriented programs into the ABS toolchain.

Integration of family-based type checking. The functional and object-oriented sublanguages of ABS interact only on the level of expressions, and variable declarations. An expression in the object-oriented sublanguage may use a function or constructor from the functional sublanguage, and any field, method parameter or variable declaration may refer to an ADT. Consequently, in addition to the already existing analysis that check the two sublanguages independently, we must also ensure the following properties:

(1) expressions in functions that use objects must be well typed;
(2) functional expressions in methods must be well typed;
(3) all classes and interfaces referred in an ADT or a function must be present in all variant where the ADT or function occurs; and
(4) all ADT, type constructors and function referred in an interface or class must be present in all variant where the interface or class occurs.

The points 1 and 2 can be ensured by combining the partial typing analysis of the two sublanguages: by combining together the class table from the object-oriented sublanguage and the $\Gamma_{\texttt{Spl}}$ table described in this paper, we can ensure that every usage of an element declared in the resulting table is correct w.r.t. its declaration.

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

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

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

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

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

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

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

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

$$\frac{\overline{d,T\vdash\texttt{KAO}_i:\Phi_i}}{d\vdash \textbf{modifies}\ \textbf{data}\ T\ \{\ \overline{\texttt{KRO}}\ \texttt{KAO}_{n+1}\ \ldots\ \texttt{KAO}_m\ \} : \bigwedge_{1\le i\le m}\Phi_i}\ \text{D:15} \qquad \frac{\overline{d,T.\mathsf{K}\vdash v_i:\Phi_i}}{d,T\vdash\textbf{adds}\ \mathsf{K}(v_1,\ldots,v_n):\bigwedge_{1\le i\le n}\Phi_i}\ \text{D:16} \qquad \frac{d\vdash\texttt{FF}:\Phi}{d\vdash\textbf{adds}\ \texttt{FF}:\Phi}\ \text{D:17} \qquad \frac{}{d\vdash\textbf{removes}\ \textbf{def}\ f:\textbf{true}}\ \text{D:18}$$

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

$$\frac{\rho_1=\mathsf{f}.\rho.\mathsf{l}\quad\rho_2=\Theta(\mathsf{f},\mathsf{l})\quad\overline{d,\rho_2\vdash\texttt{BMO}_i:\Phi_i}\quad\overline{d,\rho_2\vdash\texttt{BAO}_i:\Phi_i}\quad\overline{d,\rho.l\vdash\texttt{CaseO}_i:\Phi_i}}{d,\rho\vdash\textbf{modifiesCase}\ \mathsf{l}\ \{\ \overline{\texttt{BRO}}\ \texttt{BMO}_1\ \ldots\ \texttt{BMO}_m\ \texttt{BAO}_{m+1}\ \ldots\ \texttt{BAO}_o\ \texttt{CaseO}_{o+1}\ \ldots\ \texttt{CaseO}_p\ \} : \bigwedge_{1\le i\le p}\Phi_i}\ \text{D:21} \qquad \frac{d,\rho.p\vdash e:\Phi}{d,\rho\vdash\textbf{modifies}\ p\Rightarrow e:\Phi}\ \text{D:22} \qquad \frac{d,\rho.p\vdash e:\Phi}{d,\rho\vdash\textbf{adds}\ p\Rightarrow e:\Phi}\ \text{D:23}$$

Figure 13: Dependency Analysis

Most of the points 3 and 4 are guaranteed by the dependency analysis of the two sublanguages. The only missing part is checking that the classes and interface subtyping relation hold during the dependency analysis of the functional sublanguage. This part can be ensured by extending the rules in Figure 13 to include the subtyping relation as it is already done in the dependency analysis of the object-oriented sublanguage.

Syntactical and Transformational Considerations. On a more syntactic level, we point out that we require only to extend the current implementation of ABS with the ability to label `case`-expressions. Furthermore, the DOP operations are analogous for both sublanguages, meaning that the integration of variant generation would be purely on (named) subtrees in the AST and must not introduce new concepts beyond the ability to refer to other kinds of sub-ASTs.

8 RELATED WORK

The notions of type-safety and type-uniformity for Delta-F^{2}ABS, given in Sect. 5, are inspired by analogous notions for DOP of SPLs of Java-like programs [2], which in turn are inspired by analogous notions for FOP of SPLs of Java-like programs [22]. Also the associated family-based analysis for Delta-F^{2}ABS, given in Sect. 6, are inspired by corresponding family-based analysis for DOP of SPLs of Java-like programs [7, 8] and for FOP of SPLs of Java-like programs [10, 22]. However, the development of the case study illustrated in Sect. 2 required to devise a way to specify deltas on function definitions that are more fine-grained than the deltas on class definitions considered in the mentioned works on FOP/DOP. Namely, to define and analyze the `modifiesCase` operation – a delta-operation that works synergistically with the delta-operations on on ADTs by allowing developers to modify any occurrence of a case-expression in the body of a function.

The *Variational Lambda Calculus (VLC)* [3, 4] is a conservative extension of the implicitly typed λ-calculus with constructs for introducing and organizing static variability in λ-calculus expressions. The VLC is inspired by the more abstract *Choice Calculus* [12], which (like the VLC) is a fundamental representation of software variation designed to serve as a foundation for theoretical research in the field (indeed, as pointed out in [4], VLC can be understood as the instantiation to λ-calculus of the Choice Calculus). In [3, 4] a type assignment system for VLC is defined by introducing the notion of variational types (which lift variations from λ-calculus expressions to types), and a variational type inference algorithm that is proved correct, complete and most general. Namely, the algorithm (which is an extension of the algorithm $\mathcal{W}$ for λ-calculus [6])

takes a VLC expression *e* and returns a variational type. A successfully inferred variational type indicates that all variants of the *e* are type correct, and the type of each variant *v* of *e* can be obtained by applying to the variational type of *e* the same configuration choices applied to *e* for obtaining *v*. The paper [4] also briefly outlines with a few examples how to add to VLC sum types (in the formulation presented in [18]), which are the basis for ADTs. The main differences between Delta-F^2ABS and VLC are as follows: (1) VLC expresses variability by an annotative approach, while Delta-F^2ABS considers DOP, which is a transformational approach (see, e.g., [20, 23] for a classification of SPLs implementation approaches); (2) VLC variants are *implicitly* typed λ-calculus expressions, while Delta-F^2ABS variants are expressed in F^2ABS, which can be understood as an *explicitly* typed λ-calculus with function definitions and ADT definitions; and (3) VLC has been designed looking at theoretical research [3, 4], while (although aimed at providing a rigorous foundation) Delta-F^2ABS has been designed looking at practical application in industrial modeling scenarios.

Lambda VL [21] is a lambda calculus extended to support multiple versions in one program and equipped with proper notion of type safety. As pointed out in [21], the DOP approach is complementary to the line of research that led to the development of Lambda VL, which aims to enable programmers to more freely combine and control programs of different versions in a single code.

9 CONCLUSION AND FUTURE WORK

We have proposed and formalized DOP for SPLs of functional programming with ADTs, together with family-based analyses to check whether an SPL satisfies certain well-formedness and conditions and whether all variants can be generated and are well-typed programs. Although the proposed techniques apply to higher-order functional languages in general (cf. footnote 3 in Sect. 3), our presentation is tailored to the functional sublanguage of the multi-paradigm modeling language ABS, which already supports DOP for its class-based object oriented sublanguage. In future work, we would like to extend our formalization to cover useless-operation absence analysis, to merge it with the existing formalization of DOP for the object-oriented sublanguage of ABS [7], to implement it, and to integrate it into the ABS toolchain (cf. Sect. 7).

ACKNOWLEDGMENTS

We thank the anonymos SPLC 2023 reviewers. This publication is part of the project NODES which has received funding from the MUR – M4C2 1.5 of PNRR with grant agreement no. ECS00000036. The work was also partially supported by Italian PRIN projects CommonWears (2020HCWWLP) and T-LADIES (2020TL3X8X), by the Ateneo/CSP ex-post 2020 project NewEdge, and by the RCN projects PeTWIN (294600) and SIRIUS (237898).

REFERENCES

[1] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. 2004. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering* 30 (2004), 355–371. https://doi.org/10.1109/TSE.2004.23
[2] Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. 2013. Compositional type checking of delta-oriented software product lines. *Acta Informatica* 50, 2 (2013), 77–122. https://doi.org/10.1007/s00236-012-0173-z
[3] Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2012. An Error-Tolerant Type System for Variational Lambda Calculus *(ICFP '12)*. Association for Computing Machinery, New York, NY, USA, 29–40. https://doi.org/10.1145/2364527.2364535
[4] Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2014. Extending Type Inference to Variational Programs. *ACM Trans. Program. Lang. Syst.* 36, 1, Article 1 (mar 2014), 54 pages. https://doi.org/10.1145/2518190
[5] Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. 2010. Variability Modelling in the ABS Language. In *FMCO (LNCS, Vol. 6957)*. Springer, 204–224. https://doi.org/10.1007/978-3-642-25271-6_11
[6] Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico) *(POPL '82)*. Association for Computing Machinery, New York, NY, USA, 207–212. https://doi.org/10.1145/582153.582176
[7] Ferruccio Damiani, Reiner Hähnle, Eduard Kamburjan, Michael Lienhardt, and Luca Paolini. 2023. Variability modules. *Journal of Systems and Software* 195 (2023), 111510. https://doi.org/10.1016/j.jss.2022.111510
[8] Ferruccio Damiani and Michael Lienhardt. 2016. On Type Checking Delta-Oriented Product Lines. In *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings (LNCS, Vol. 9681)*. Springer, 47–62. https://doi.org/10.1007/978-3-319-33693-0_4
[9] Ferruccio Damiani, Michael Lienhardt, Bruno Maugars, and Bertrand Michel. 2022. Towards a Modular and Variability-Aware Aerodynamic Simulator. In *The Logic of Software. A Tasting Menu of Formal Methods (Lecture Notes in Computer Science, Vol. 13360)*. Springer, 147–172.
[10] Benjamin Delaware, William R. Cook, and Don Batory. 2009. Fitting the Pieces Together: A Machine-checked Model of Safe Composition. In *ESEC/FSE* (Amsterdam, The Netherlands). ACM, 243–252. https://doi.org/10.1145/1595696.1595733
[11] Richard A. Eisenberg, Joachim Breitner, and Simon Peyton Jones. 2018. Type Variables in Patterns. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/3242744.3242753
[12] Martin Erwig and Eric Walkingshaw. 2011. The Choice Calculus: A Representation for Software Variation. *ACM Trans. Softw. Eng. Methodol.* 21, 1, Article 6 (dec 2011), 27 pages. https://doi.org/10.1145/2063239.2063245
[13] A. Igarashi, B. Pierce, and P. Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM TOPLAS* 23, 3 (2001), 396–450. https://doi.org/10.1145/503502.503505
[14] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2010. ABS: A Core Language for Abstract Behavioral Specification. In *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*. 142–164.
[15] Eduard Kamburjan, Reiner Hähnle, and Sebastian Schön. 2018. Formal Modeling and Analysis of Railway Operations with Active Objects. *Science of Computer Programming* 166 (Nov. 2018), 167–193. https://doi.org/10.1016/j.scico.2018.07.001
[16] Michael Lienhardt and Dave Clarke. 2012. Conflict Detection in Delta-Oriented Programming. In *ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I (LNCS, Vol. 7609)*. Springer, 178–192. https://doi.org/10.1007/978-3-642-34026-0_14
[17] Reza Mauliadi, Maya R. A Setyautami, Iis Afriyanti, and Ade Azurat. 2017. A platform for charities system generation with SPL approach. In *Proc. Intl. Conf. on Information Technology Systems and Innovation (ICITSI)* (Bandung, Indonesia). IEEE, New York, NY, USA, 108–113. https://doi.org/10.1109/ICITSI.2017.8267927
[18] Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.
[19] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond (SPLC 2010) (LNCS, Vol. 6287)*. 77–91. https://doi.org/10.1007/978-3-642-15579-6_6
[20] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. 2012. Software diversity. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 477–495. https://doi.org/10.1007/s10009-012-0253-y
[21] Yudai Tanabe, Luthfan Anshar Lubis, Tomoyuki Aotani, and Hidehiko Masuhara. 2022. A Functional Programming Language with Versions. *Art, Science, and Engineering of Programming* 6, 1 (2022). https://doi.org/10.22152/programming-journal.org/2022/6/5
[22] Sahil Thaker, Don Batory, David Kitchin, and William Cook. 2007. Safe Composition of Product Lines. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering* (Salzburg, Austria) *(GPCE '07)*. ACM, New York, NY, USA, 95–104. https://doi.org/10.1145/1289971.1289989
[23] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1, Article 6 (2014), 45 pages. https://doi.org/10.1145/2580950
[24] Peter Y. H. Wong, Nikolay Diakov, and Ina Schaefer. 2012. Modelling Distributed Adaptable Object Oriented Systems using HATS Approach: A Fredhopper Case Study (invited paper). In *2nd Intl. Conf. on Formal Verification of Object-Oriented Software, Torino, Italy (LNCS, Vol. 7421)*, Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov (Eds.). Springer, 49–66. https://doi.org/10.1007/978-3-642-31762-0_5