CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A type-driven approach for sensitivity checking with branching

Master's thesis in Computer science and engineering

Daniel Freiermuth

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Master's thesis 2023

A type-driven approach for sensitivity checking with branching

Daniel Freiermuth

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

A type-driven approach for sensitivity checking with branching

Daniel Freiermuth

© Daniel Freiermuth, 2023.

Supervisor: Alejandro Russo, Chalmers
Advisor: Marco Gaboardi, DPella
Examiner: David Sands, Chalmers

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

A type-driven approach for sensitivity checking with branching

Daniel Freiermuth
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Differential Privacy (DP) is a promising approach to allow privacy preserving statistics over large datasets of sensitive data. It works by adding random noise to the result of the analytics. Understanding the sensitivity of a query is key to add the right amount of noise capable of protecting privacy of individuals in the dataset. The domain-specific language Spar[1] implements a programming language that uses the type system to automatically track the sensitivity of queries in Haskell. Queries can be constructed from basic operations in an EDSL. The operations' impact on sensitivity need to be well-known and made explicit at type-level. Spar lacks branching operations. In general branching is a discontinuous operation, so the sensitivity of the whole branch might not be bounded. Due to this reason, most languages that track sensitivity do not provide branching as a basic operation. We introduce a modular and type-driven branching operation that checks for continuity at compile-time. It is implemented in Template Haskell and thus operates on the syntax of the condition and bodies of the branches. To demonstrate our approach, we provide basic examples common in literature. Additionally, we also provide the implementation of more sophisticated operations such as Mergesort. We develop requirements under which the use of our branching operator is sound.

Keywords: Computer, science, computer science, thesis, differential privacy, type system, sensitivity, branching.

Acknowledgements

This thesis would not look the same without the constant stream of feedback I got from my supervisor Alejandro Russo. I also want to thank Marco Gaboardi for mathematical discussions and my examiner David Sands for his insightful remarks. Thank you, Elisabeth and Augustín for providing me with tips and ideas and sharing your perspective on whatever topic I needed help with. Thank you, Erik for your considerate and patient in-depth feedback. Thank you Beata, my opponent, for sharing this journey and for the fruitful feedback sessions. Finally, I want to thank all my family and friends who supported me during this intense time and had an open ear and heart whenever I needed it. Without you, this thesis would not have happened.

Daniel Freiermuth, Gothenburg, 2023-08-01

Contents

1

Introduction

Imagine having large amounts of sensitive data that is interesting for research. Data like a hospital or a health insurance have. It is desirable to allow researchers to perform analyses, called queries, on this dataset. The problem is that it might be possible to regain insights on specific data points, i.e. data subjects. There should be a way to allow analyses on the data without sacrificing the individuals' privacy.

Typically, privacy can be preserved by distorting the analysis. Differential Privacy (DP)[2] is one method for doing so. It works by, first, keeping track of how sensitive a given analysis is to the impact of individuals. Then calibrated noise is added to the result of the analysis. The idea is that the noise outweighs an individual's impact. It is particularly interesting because, in contrast to other methods, it does not rely on obscurity of the process.

In order to protect an individual's privacy, it is important to understand the impact of an individual to the result. Sensitivity describes how much the output changes, given a specific change of the input. In our context, this means how much the result of a query can change given the modification of an individual.

Consider a dataset with 10000 people. In order to query the average weight privacy-preservingly, it is necessary to understand the sensitivity of this analysis. Given that people normally weight between 30 and 200 kg, the result will change at most by 0.0017 kg when replacing a person. Adding noise sampled from a normal distribution with variance σ=0.01 kg greatly blurs an individual's impact while leaving the result quite usable. Note that we made this judgement without knowing the data. It is sufficient to know the sensitivity of the query and then use knowledge of value ranges in the dataset in order calculate the necessary magnitude of noise. Sensitivity tells us how the range of possible values is scaled by the query. That means, it is possible to understand the sensitivity of a query statically without knowing the data.

This leads to the idea of automatically inferring the sensitivity of a query. Different approaches[3], [4] have been taken creating Domain Specific Languages and calculi for expressing queries. Those languages track the queries' sensitivities in the typing information using linear[3] and dependent[4] types.

A similar approach is Spar[1] which presents an Embedded Domain Specific Language (EDSL) for expressing queries. It improves on the previous solutions in that it is implemented in Haskell and tracks the sensitivity of a query constructed in the Haskell type system without the need of more exotic type features. Queries can be

constructed by combining simple primitive atomic queries. The possible ways of combination are limited and the sensitivity of the combination is derived from the sensitivity of the combined queries.

For example, our query from before could look like this.

```
sum_weight rows = sum $ map (\row -> weight row) rows
```

Here, the database is passed as `rows` parameter. `weight` is the atomic query giving us access to the weight data, which is summed with the `sum` operation. After dividing the result by 10000 (the number of individuals in the dataset), the average is calculated. This query has sensitivity 1, which means that a change in an individual's weight translates into the same change in the result.

Formally, sensitivity means *Lipschitz continuity*.

Definition 1. *A c-Lipschitz continuous function f is characterized by*

$$\forall x, y : |f(x) - f(y)| \leq c \cdot |x - y|$$

In the context of DP, it is an upper bound on the change of the result of a query if one individual's data is added, changed or removed. In the definition of Lipschitz continuity, f matches the query while x and y express different datasets. We call a function *insensitive* if there cannot be a real upper bound of its sensitivity.

Unfortunately, not all operators produce queries which are sensitive. Consider this function for calculating the average BMI which uses division.[1]

```
sum_bmi rows = sum $ map (\row -> weight row / ((height row) ^ 2)) rows
```

Without knowing more about the physicalities of humans, a change in the weight of a very small individual with a height near zero results in a big change of its BMI and thus the result. A change in an individual's weight may be scaled by any degree in the result. This query's sensitivity is not bound and there is no noise magnitude that will nicely outweigh an individual's impact.[2]

This shows why the set of possible combinators in Spar is limited. Division and multiplication are not supported.

Another operation omnipresent in many algorithms in the form of if-then-else constructs is branching. Branching means checking a condition and evaluating one of two expressions (called branches) depending on whether the condition is fulfilled.

Unfortunately, branching also introduces the possibility to create queries of unbounded sensitivity. The following query counts individuals heavier than 100 kg. A small difference in an individual's weight may translate into a difference of 1 in the result. Again, there is no scaling factor that connects the difference in the input

[1] Given the height h and mass m of a person, its BMI is defined by $\frac{m}{h^2}$

[2] Added to this, in the current implementation, Spar does keep track of sensitivities, but not value ranges. Values range knowledge is later combined with the calculated sensitivity to determine the required noise.

to the change in the result. This query is unbounded and shows why branching is neither supported.

```
countHeavy rows =
 sum $ map (\row -> if weight row > 100 then 1 else 0) rows
```

This becomes clear for two similar lists of weights l1=[30, 100, 130] and l2=[30, 100.1, 130]. The distance between the lists of weights is given by norm L1, i.e. the summed pointwise distances of the elements. The lists are 0.1 apart. After applying `countHeavy`, the results are 1 apart: `countHeavy l1 = 1`, `countHeavy l2 = 2`. The distance in the input is scaled by 10. By replacing the element 100.1 by 100.01 in l2, we can increase the scaling factor to 100. Similarly, it is possible to show every possible scaling factor.

Yet, there are interesting sensitive queries that could be implemented using branching. Imagine a `max` function, returning a maximal value in a list. A change in an individual's weight will, at most, result in the same change of the result.

```
max_weight rows = max $ map (\row -> weight row) rows

max [r]    = r
max (r:rs) = if r > max rs then r else max rs
```

In literature, many of the previous work supporting sensitivity calculations (e.g., Fuzz [3], DFuzz [4], Solo [5]) do not support sensitive branching natively. Instead, an additional combinator `cswp` is provided. With cswp, interesting algorithms that use branching, can be rewritten without branching but using the cswp operator. `cswp` computes the compare-and-swap function

$$\text{cswp}(x, y) = \begin{cases} (x, y) & x > y \\ (y, x) & \text{otherwise} \end{cases}$$

With this combinator in place, it is possible to define the `max` function.

```
max_weight rows = max $ map (\row -> weight row) rows

max [r]    = r
max (r:rs) = let (b,s) = cswp(r, max rs) in b
```

Sorting is another sensitive operation that needs branching. Slow sorting algorithms like Bubblesort can be expressed with `cswp` but faster ones like Mergesort cannot. This is because, the algorithm cannot be expressed on sorting pairs.

For making queries differentially private straightforwardly, it would be desirable to allow arbitrary branchings in a framework for sensitive computations. That would allow direct translations of existing algorithms and alleviates the need of providing special purpose operators as cswp.

1.1 Goal

The goal of this thesis is to extend the Spar framework by a branching operator that allows branchings. As seen above, not every branching is sensitive. Compile-time checks will be performed before accepting a branching. This will remove the need for providing special-purpose workarounds like cswp. Additionally, it will allow defining additional queries which cannot be written with cswp like Mergesort (see Chapter 7.1).

This will be achieved by combining ideas from Chaudhuri *et al.*[6] with Spar. Chaudhuri *et al.* present a calculus for reasoning about sensitivity of imperative programs that may branch. This calculus considers a branching to be sensitive if two conditions are fulfilled.

Firstly, both branches must be sensitive on their own. The calculus is applied recursively to both branches. If for both branches a sensitivity is derived, the first condition is fulfilled.

Secondly, the branching must be continuous. Continuity as a mathematical concept was already presented in the notion of Lipschitz-Continuity. In the context of branching this corresponds to the intuition that the result must not *jump* when going from one branch to the other. This means that both branches must be equivalent on the decision boundary. We say that the branches *agree* on the decision boundary. The decision boundary is the set of variable assignments that are arbitrarily close to other assignments that make the condition flip.

With those two conditions, the branching is sensitive with the larger of the branches' sensitivity. Let's understand those condition with some examples.

Consider the following branching. For values of `x` greater zero, a small change of `x` results in a doubling of this change in the result. When transitioning from one branch to the other, i.e. when `x` flips sign, the result is zero in both branches. If considering both branches as function of `x`, it is easy to see that they are sensitive with sensitivity 2 and 1, respectively. The decision boundary contains the single assignment (x=0). For all values on the boundary, i.e. for x=0, both branches evaluate to 0. So both branches are equivalent on the boundary. This function is sensitive with sensitity 2.

```
if x > 0 then
    2 * x
else
    x
```

Here are counter-examples of branchings that are not sensitive. The first example consists of two sensitive branches, but the branches do not agree on the decision boundary x=0.

```
if x > 0 then
    -x + 1
else
    x
```

In order to show that its sensitivity is unbounded, consider two inputs $x_1 = 0$ and $x_2 = 0.1$ resulting in the outputs 0 and 0.9. By moving $x_2 > 0$ closer to zero, the sensitivity becomes arbitrarily large.

In the second example the condition is altered. The decision boundary is now x=1. The branches do not agree on this boundary, the branching is not continuous, and we do not consider it sensitive.

```
if x > 1 then
    2 * x
else
    x
```

Again, consider two inputs $x_1 = 1$ and $x_2 = 1.1$ with results 1 and 2.2. Moving $x_2 > 1$ closer to one generates arbitrarily sensitivities.

The third example contains shows a branching that is continuous, but the first branch is insensitive. So the whole branching is insensitive.

```
if x > 0 then
    x * x
else
    x
```

Here, consider two inputs $x_1 > 0$ and $x_2 = x_1 + 1$ with results x_1 and x_1^2. Increasing x_1 increases the distances in the results arbitrarily while the distance in the inputs stays the same.

Our goal is that the operator to be constructed allows the first example while rejecting the other three examples at compile-time.

1.2 Approach

The branch operator will be implemented as a GHC plugin that analyses the branching at compile-time. The analysis will be done along the two conditions presented in the previous section. As the goal is to extend Spar, its types for tracking sensitivity can be reused. This will help us reason about the sensitivity of the branches. It remains to check for agreement of the branches on the decision boundary. Our plugin will thus analyze the condition, calculate the decision boundary and check whether the branching is continuous. For calculating the decision boundary an SMT solver is used.

With our plugin, we will implement the cswp function as a function on top of Spar instead as a black-box primitive. Additionally, we implement Mergesort as an algorithm which cannot be described with cswp.

2

Background

This section presents the main concept, tools and techniques used in this thesis. Most of them were already dropped in the introduction. This chapter gives more context to all of them.

2.1 Differential Privacy (DP)

Differential Privacy is a method for querying sensitive data in a privacy-preserving way. It was developed by Dwork, McSherry, Nissim, *et al.*[7], [8].

Differential Privacy is a property of an algorithm or query. If it is fulfilled, an individual's impact to the result can hardly be detected. In this paper the formulation by Kifer and Machanavajjhala[9] will be used.

Definition 2 (Bounded Differential Privacy[9])**.** *A randomized algorithm A satisfies bounded ϵ-differential privacy if $P(A(D_1) \in S) \leq e^{\epsilon} \cdot P(A(D_2) \in S)$ for any set S and any pairs of databases D_1, D_2 where D_1 can be obtained from D_2 by changing the value of exactly one tuple.*

Consider a dataset D_1 and a change in an individual's data yielding D_2. Let A be a randomized algorithm. The behavior of A on D_1 can be described by $P(A(D_1) \in S)$ for every set S. Differential Privacy tells us that an ϵ-differentially private randomized algorithm A will hardly change its behavior, i.e. its probabilities. In particular: $P(A(D_1) \in S) \leq e^{\epsilon} \cdot P(A(D_2) \in S)$ with ϵ typically being a small positive number. The parameter ϵ is often referred to as the privacy parameter and is used to control the strength of the privacy requirement.

Note that privacy is not binary here. An algorithm is not either privacy-preserving or leaking insights. It is leaking insights with some probability e^{ϵ}. For large ϵ the algorithm probably would not be considered privacy-preserving in another context.

The beauty of DP lies in its properties. DP does not require the algorithm to be obscure in order to guarantee privacy. The mathematical formulation allows considering the composition of several queries analysing the resulting privacy.

DP is popular because of various claims that have been made about it: [9]

- It makes no assumptions about how data is generated.

- It protects an individual's information (even) if an attacker knows about all other individuals in the data.
- It is robust to arbitrary background knowledge.

However, those claims are not generally true and were critically analyzed in the same paper [9].

There are more notions of DP. Besides bounded DP, there is unbounded DP and formulations with more privacy parameters. The main purpose of this section is to get an idea in which context branching with sensitivity-checking may be applied. It is not necessary to understand the different notions and technicalities of DP for following this thesis.

2.1.1 Making sensitive queries differentially private

After understanding why differentially private algorithms are interesting, the question arises how to obtain them. It turns out that it is possible to turn sensitive algorithms into differentially private ones.

Turning sensitive queries into differentially private queries was pioneered by McSherry[2]. In this paper sensitivity is referred to as transformation stability. This approach was further developed by Reed and Pierce[3] and led to the *laplace mechanism* as formulated in [3]. L_k describes the Laplace distribution with parameter k. Its probability density function is $p(x) = \frac{1}{2k} \cdot \exp(-|x|/k)$.

Lemma 1 (Laplace mechanism[3]). *Suppose $f : \texttt{db} \to R$ is c-sensitive. Define the random function $q : \texttt{db} \to R$ by $q = \lambda b.f(b) + N$, where N is a random variable distributed according to $L_{c/\epsilon}$. Then q is ϵ-differentially private.*

This mechanism works straightforwardly. The derived differentially private query q executes the original sensitive query f and then adds calibrated noise N in order to blur an individual's impact. The added noise is scaled according to the privacy parameters and sensitivity of f such that the noise outweighs (according to the privacy parameter) an individual's impact.

2.2 Spar-calculus

Spar, first presented in [1], is a Haskell library for facilitating the calculation and inference of sensitivity. Sensitivity is tracked on the type level. The central type is `Sen`. The function `f :: Sen 2 Int Int` is a function with sensitivity 2 that converts an `Int` into an `Int`. With Spar it is possible to automatically infer this type.

```
f :: Sen 2 Int Int
f x = x .+ x
```

Under the hood this works by tracking how a change in the input translates into some change in the result of the function. This aligns with the intuition of sensitivity developed in the introduction chapter. This works by introducing another type

`Dist d a`. A value `v :: Dist d a` denotes a pair of values of type `a` with distance at most `d`. For a type `a` with distance metric $\|\cdot\|$, we identify the type `Dist d a` with the mathematical set $\{(x, y) \in \mathtt{a} \times \mathtt{a} \mid \|x - y\| \leq \mathtt{d}\}$. In the following we will refer to values of type `Dist d a` as Spar-values of type `a`. We think of such a pair (x, y) as two inputs or outputs of a function in two separate executions. To distinguish those pairs of executions from other pairs, we will write $x \sim y$ instead of (x, y) later and $x \sim_d y$ to denote that x and y are d apart.

It is now possible to translate the intuition of sensitive functions translating changes into our types. Given a pair of two values $x \sim_d y$, we can think of it as obtaining y by changing x by d. A 1-sensitive function f does translate a change in the input to at most the same change in the output. So $f(x)$ will be at most d away from $f(y)$. Writing this with our tuple notation: $f(x) \sim_d f(y)$. Sensitivity can be described by how f affects the distance of a pair when passing the pair through f. A k-sensitive function g translates a pair $x \sim_d y$ into another pair $g(x) \sim_{k \cdot d} g(y)$. This can be expressed in Spar' `Dist` type.

```
f :: Dist d Int -> Dist (2 * d) Int
f x = x .+ x
```

This is exactly how the type `Sen` is derived:

```
type Sen k t1 t2 = forall d. Dist d t1 -> Dist (k*d) t2
```

All values are encapsuled in Spar-values. Those values can be combined and processed in limited ways by applying primitives. Those primitives' typing information indicates how they affect the distances. Observing a whole query's typing information, the query's sensitivity can be concluded.

In the following the relevant interactions with Spar-values are presented. Figure 2.1 shows the relevant API.

Spar provides a basic arithmetic framework. `Dist`-values can be constructed from integer literals via `constant` or `number`. Addition and subtraction of Spar-values of type `Int` is provided by `(.+),(.-)`. The underlying constructors of `Dist d Int` are hidden, so that the actual values remain obscure. This also effectively prevents branching on sensitive values.

```
a = constant 4 :: Dist 0 Int
b = number 3 :: Dist 3 Int
a .+ b :: Dist 3 Int
```

Two Spar-values can be combined in a tuple via `(:*:)`. Note that resulting type discards detailed information about the radii of the tuple's elements. Only the resulting distance is stored. This corresponds to the L_1-norm. Other norms can be applied at Spar, like L_∞, but that requires modifying the type signature of `(:*:)`. For simplicity, we stick with L_1-norm. Note that, by having pattern synonyms, we can deconstruct pairs.

```
pair = a :*: b :: Dist 3 (Int, Int)
let a' :*: b' = pair
```

```
data Dist (d :: Nat) a -- abstract
type Sen (k :: Nat) a b = forall n. Dist n a -> Dist (k*n) b.

-- Integers
constant :: Int -> Dist 0 Int
number :: forall n. Int -> Dist n Int
(.+),(.-) :: Dist d1 Int -> Dist d2 Int -> Dist (d1+d2) Int

-- Pairs
pattern (:*:) :: Dist d1 t1 -> Dist d2 t2 -> Dist (d1+d2) (t1,t2)
{-# COMPLETE (:*:) #-}

-- Lists
pattern Nil  :: Dist d (Vec 0 a)
pattern (:>) :: Dist d1 a -> Dist d2 (Vec l a)
                -> Dist (d1+d2) (Vec (l+1) a).
{-# COMPLETE Nil, (:>) #-}

-- Execution
run :: Sen k a b -> a -> b
```

Figure 2.1: The relevant Spar-API.

Support for vectors is provided by `Nil` and `(:>)`. Again only the total distance is stored and by exporting pattern synonyms, lists can be deconstructed. This makes vectors similar in a sense that a vector `v::Dist d (Vec n t)` of fixed length n can be replaced by the n-tuple `p :: Dist d (t,(t, (t, …)))`.

```
vec = 3 :> 4 :> Nil :: Dist 3 (Vec 2 Int)
let hd :> tl = vec
```

By this construction, every type comes with a distance metric associated. This metric is implicitly described by the types' constructors.

Finally, a `run` function for sensitive functions is provided. This function forgets the type level sensitivity information and executes a query as the last step. Typically, it will be called by the same trustworthy instance adding the noise that is calibrated by the type level information. Particularly, it is not aligned with its purpose to use `run` to build a query.

With this it is possible to write easy queries which sensitivity are inferred by the type system. Those then can be `run` against the data and the appropriate noise can be added.

2.3 Compare and Swap (cswp)

While introducing and refining our branch operator, we will use the compare-and-swap function (cswp) function as a running example. It was already presented in the introduction.

$$\text{cswp}(x, y) = \begin{cases} (x, y) & x > y \\ (y, x) & \text{otherwise} \end{cases}$$

In the literature[3], [4] it serves as a primitive to work around limitations of not having primitive branching.

With cswp, it is easy to implement derived functions like min and max but also primitive sorting algorithms like Bubblesort as also done by others([3]). Also the graph algorithms of Dijkstra and Bellman-Ford as sketched by Chaudhuri, Gulwani, and Lublinerman[6] can be implemented with cswp.

```
min a b = let (bigger :*: smaller) = cswp (a :*: b) in smaller
max a b = let (bigger :*: smaller) = cswp (a :*: b) in bigger
relu x = let (bigger :*: smaller) = cswp(x, constant 0) in bigger
abs x = let (bigger :*: smaller) = cswp(x, constant 0 .- x) in bigger

bubbleSort Nil = Nil
bubbleSort list =
    let (biggest :> remaining) = bubbleUp list
    in (biggest :> bubbleSort remaining)

    where
        bubbleUp Nil = Nil
        bubbleUp (e :> Nil) = (e :> Nil)
        bubbleUp (head :> tail) =
            let (biggestInTail :> remainingTail) = bubbleUp tail
                (biggest :> other) = cswp (biggestInTail :*: head)
            in (biggest :> other :> remainingTail)
```

2.4 Haskell technicalities

SimpleSMT is a library providing an API to interact with SMT solvers. The Z3 solver is supported.

Haskell allows to alter the compilation at several stages using plugins. Figure 2.2 shows the Haskell compilation pipeline with possible plugins. Plugins are defined in their own Haskell module using the GHC (Glasgow Haskell Compiler) plugins API.

At source stage, plugins have access to the source code where a source plugin can alter code. No further analysis by the compiler has taken place at this point.

A type checking plugin helps the type checker to infer or unify types. It is called with expressions and types and can return typing suggestions. This is especially useful

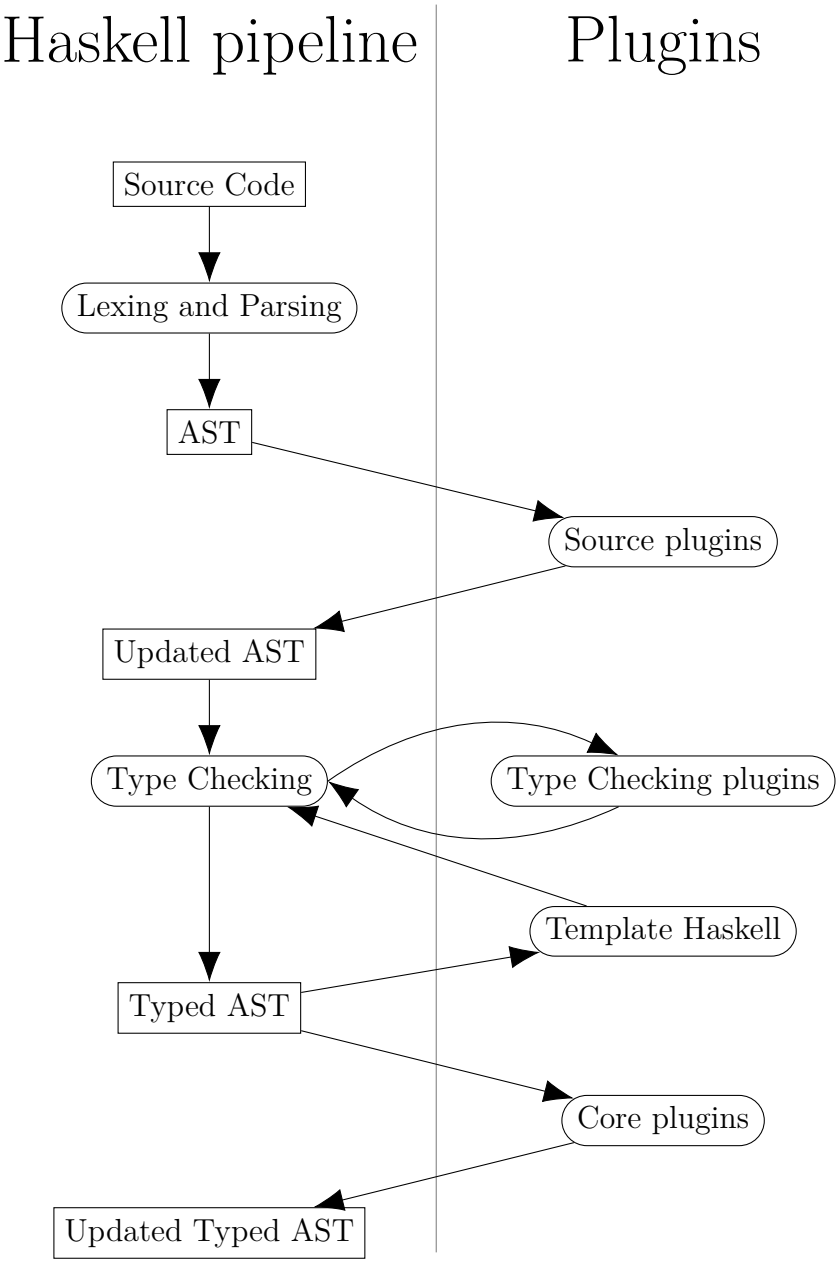

Figure 2.2: Overview of Haskell compilation and plugin stages.

where strong static guarantees are carried in types, yielding complicated calculations in types which GHC is incapable of or slow at.

Thoralf[10] is a specific type checking plugin, that helps the type checker unifying types. It is mostly a wrapper to an SMT solver. For this, some theories like arithmetic are built-in. Other custom theories can be provided. The unification challenge then translates to an SMT challenge, which is given to Z3. As an example consider the type level natural numbers `a,b,c,n1,n2,n3 :: Nat` with known constraints `(n1 + n2) :~: (a + c)` and `(c + n3) :~: b`. With Thoralf it is possible to deduce the constraint `(n1 + (n2 + n3)) :~: (a + b)`.

Core plugins work on Haskell's internal Core representation of the program. This includes type checking already done.

Template Haskell is another way to interfere with Haskell's compilation. Strictly speaking, it is not a plugin, but more a macro with Haskell code run at compile time based on a GHC extension. Such macros can take other Haskell code (well-typed parts of the AST) as argument and are expected to return Haskell code themselves. The result is then passed to the type-checker again.

2.5 Solvers

A satisfiability solver (SAT solver) is a tool to solve boolean expressions with variables. Given a boolean expression, it finds a satisfying assignment for the variables such that the expression becomes true.

The result of a SAT solver is either the signal SAT with some fulfilling assignment or UNSAT indicating that it is impossible to satisfy the expression.

2.5.1 Adding arithmetic support

SMT solvers (Satisfiability modulo theories) extend SAT solvers by also allowing predicates in some so-called theory. In our context, this will be the arithmetic theory of integers with comparisons, addition and subtraction. Besides boolean variables directly occurring in the boolean expression, predicates of the theory like $x + y > 0$ can be used in the boolean expression. With this, a second set of variables occurs: value variables in the theory (here x and y). This makes predicates as

$$x < y \vee x + y = 0 \tag{2.1}$$

available in formulas.

The logical expression is solved by first treating the predicates as unique variables and finding a satisfying assignment for those (SAT solving). The expression (2.1) from before would be translated to $p1 \vee p2$ with $p1 \equiv x < y$ and $p2 \equiv x + y = 0$. A first satisfying assignment could be $p1$ =True, $p2$ =False. Then, the soundness of predicates is checked given the theory, yielding a possible assignment of value variables or rejecting the boolean assignment. In our example, the arithmetic theory would be asked to fulfill $x < y$ while not $x + y = 0$. As there is a variable assignment

$(x = 0, y = 1)$ that solves this, this assignment could be yielded. If the found boolean assignment to $p1$ and $p2$ would be rejected, it returns to SAT solving and tries to find another SAT-assignment.

The Z3 SMT solver is the SMT solver we are using in our development.

2.5.2 Getting all solutions

If not a single fulfilling assignment, but every fulfilling assignment is to be calculated, an SMT solver can be called iteratively. After every returned SAT, the assignment is stored and a clause is added to the expression forbidding this exact assignment. By this, another assignment will be returned by the next call (if there is one).

Continuing our example, a clause would be added to the formula $(x < y \vee x + y = 0) \wedge (x \neq 0 \vee y \neq 1)$. This would probably yield $x = 0, y = 2$ and for the next iteration the formula would be updated to $(x < y \vee x + y = 0) \wedge (x \neq 0 \vee y \neq 1) \wedge (x \neq 0 \vee y \neq 2)$. In our example this process will never terminate as there are infinite assignments fulfilling the initial formula.

2.5.3 Reversing the quantification

SAT and SMT solvers yield *some* fulfilling assignment on the variables. Formally, given some expression $p \in \mathbb{V} \to \mathbb{B}$, that relates a variable assignment $v \in \mathbb{V}$ to a boolean $b \in \mathbb{B}$, they prove $\exists v \in \mathbb{V} : p(v)$ or $\neg(\exists v \in \mathbb{V} : p(v))$. If, however, not a single assignment fulfilling the expression is of interest, but whether the expression holds *for every assignment*, we can pass the negated expression to the solver. Let $p' = \neg p$. Then, the SMT outputs either $\exists v \in \mathbb{V} : p'(v)$ and p is not always true. Or the SMT solver outputs $\neg(\exists v : p'(v)) \Leftrightarrow \forall v : p(v)$.

Now, we have all the technical pieces together, we will our solution on. A new branch operator we will introduce in the next chapter. The branch operator will work on top of the Spar-calculus and use an SMT solver to check branchings at compile-time.

3

Branching on Spar-values

Our approach is to extend the existing framework Spar described in [1] with ideas from [6]. Spar defines a DSL for queries allowing basic operations (no branching) that infers the sensitivity by type checking. This is done by tracking how functions alter the distances of two values.

The work in [6] defines a calculus to reason about sensitivity of imperative programs. Given a boolean b, programs P1, P2 and a branching `if b then P1 else P2`, the main insight is that checking two conditions is enough to reason about sensitivity of the branching. The idea of this paper can be split into two main points that, once answered successfully, the sensitivity of the branch can be automatically computed.

Firstly, is the sensitivity of the branches P1 and P2 bounded? The paper [6] shows how sensitivity of an imperative program P can be computed in Lipschitz matrices that track the sensitivity between variables before and after the execution of P. Those matrices can be derived syntactically bottom-up by rules stated in [6].

Definition 3 (Lipschitz matrix[6])**.** *Let P a program with n variables $x_1, \ldots, x_n$, a Lipschitz matrix J of P is an $n \times n$ matrix, each of whose elements is a function $K : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Elements of J are represented either as numeric constants or as symbolic expressions (for example, $N + 5$), and the element in the i-th row and j-th column of J is denoted by $J(i, j)$.*

Secondly, is the branching continuous? Continuity is derived syntactically bottom-up with rules. Both branches need to be continuous on their own. Additionally, the decision boundary has to be considered. This is the set of variable assignments where the condition `b` flips. An overestimating condition for values on the decision boundary is calculated syntactically. With this, the Z3 SMT solver is employed to reason about equivalence of "straight-line program fragments"[6, p. 112]. If both branches are

```
f x = if x >= 0 then
          x
      else
          2x
```

Figure 3.1: A small sensitive Haskell function.

equivalent on the decision boundary, the branching is considered continuous.

Taking both points together, sensitivity of the branching is calculated. The resulting Lipschitz matrix is defined by the maximum of the derived matrices for the branches and thus overapproximates the sensitivity.

Now, let us combine the ideas of both papers, namely [1] and [6]. For this, we replace tracking sensitivity using matrices by tracking distances using types. If we can require `P1` and `P2` being of some type `Sen k a b` and `Sen k' a b`, we already calculated their sensitivity. If we can additionally show agreement on the decision boundary, we can conclude that distances will grow maximally by `max k k'` in the branching.

Consider the function in Figure 3.1. The Lipschitz matrices occurring are 1x1-dimensional since there is only one variable. Let's go bottom up: The then-branch's Lipschitz matrix is $J_1 = (1)$ and denotes that the result changes at most as much as the input. The else-branch's Lipschitz is $J_2 = (2)$ showing that the results depends on the input two times. To the whole branching the Lipschitz matrix $J = (2)$ is assigned as the elementwise maximum of both branches' matrices.

Consider two different inputs $\|x1 - x2\| \leq d$. Three cases can occur:

x1, x2 >= 0 If both inputs are greater or equal than 0, the distance of the outputs will stay the same. This corresponds to J_1.

x1, x2 < 0 If both inputs are smaller than 0, the distance of the outputs doubles. This corresponds to J_2.

x1 < 0 <= x2 , without loss of generality (w.l.o.g.). If one input is greater or equal and the other is smaller than 0, the distance will still double at most. This is because both branches agree at the decision boundary. The only value on the boundary is $x = 0$ and $x = 0$ equals $0 = 2x$. This corresponds to the overapproximation $J = (2)$.

We can conclude that evaluating this program for two inputs, their distance will be doubled at most. That is, the sensitivity of the program in Fig. 3.1 is 2.

```
if x > 0 then
    -x + 1
else
    x
```

Looking back at the examples in Section 1.1, we see that the branches are of sensitivity 1, but do not agree on the decision boundary. So the sensitivity of the branching cannot be computed. In fact, as we have seen, it is not bounded.

In the remaining chapter, we will introduce a branch operator that checks exactly those two points. By this, we can reason that the checked branching is sensitive. Such branchings are a sound extension to Spar. Easy examples like the absolute value function will be defined as well as the more involved cswp function.

```
f x = $(branch
            [| … |]
            [| x >= 0 |] -- condition
            [| x |]      -- then clause
            [| 2x |]     -- else clause
      )
==> evaluates to
f x = if run id x >= 0
      then x
      else 2x
```

Figure 3.2: The function from Figure 3.1 for Spar. The argument `[|…|]` will be explained later in Section 4.2.

3.1 Introducing the branch operator

Reconsider the function `f` in Figure 3.1. It does not typecheck for Spar-values. The problem is that we cannot compare Spar-values in the condition without running them. Yet, constructing Spar-values based on the result of running other Spar-values breaks the sensitiviy promise:[1]

```
mul :: Sen 1 Int Int
mul x = let x' = run id x in number (x' * x')
```

This function typechecks. But given two slightly different inputs, the outputs can change a lot: Consider $\hat{x} \in \mathbb{N}$ and `x :: Dist 1 Int` where we interpret `x` as two values $\hat{x} \sim \hat{x} + 1$. Function call `mul x` evaluates to $\hat{x}^2 \sim \hat{x}^2 + 2\hat{x} + 1$. As we can see, function `mul` scales the distance of the input by more than 1. This contradicts the promises given by the types. In fact, the outputs' distance is proportional to the inputs' value and can be arbitrary far away. The function's sensitivity cannot be bounded. Using `run` enables us to convince the type system that `mul` has sensitivity 1. This is why we cannot use `run` arbitrarily.

We define a new operator `branch`, that adds branching to Spar. Figure 3.2 shows our previous example with our new operator. This operator computes (if possible) the sensitivity of branches at compile-time. The condition is passed in the second argument and the branches in the third and fourth argument, respectively. The operator is a thin wrapper around an if-statement: Finally it is replaced by an if-statement where the Spar-values are `run` in the condition. The first argument (`[|…|]`) is a technical detail; we will deal with it later.

In connection with Spar-values, we want the branch operator to only allow branchings with bounded sensitivity. As we have seen before, two conditions have to be fulfilled. Firstly, both branches need to have some sensitivity. This is facilitated by Spar: if the branches in the resulting if-then-else typecheck, we already obtain their sensitivity.

[1] Recall that `run` should only be called at the end by trustworthy actors.

Secondly, the branches have to agree at the condition-boundary. This is explicitly checked by the branch operator. The sensitivity of the resulting if-statement is bounded by the larger of the two branches' sensitivities. In section B we will formalize this reasoning and see that it holds for more general values in the branches like differences or pairs.

In order to run the analysis at compile-time, we implemented the branch operator using Template Haskell and work on the syntax of the condition and the branches. Thus, all arguments have to be surrounded by quasiquotes (`[| . . . |]`) in order to turn the content into AST-values. As for now, we assume the branch operator only taking simple conditions where variables evaluate to integers, and where conditions involve comparison and logical operators, e.g., `x < 0 || x > 10`. This simplification allows us to statically calculate the decision boundary using an external SMT solver. For all boundary values `v` temporary Haskell snippets are constructed with the condition variables set to `v`. The snippets consist of the code of both branches and the operator proceeds to compare the results. If all snippets evaluate to `True` indicating agreement of both branches on all boundary values, the check is passed and the operator evaluates to a simple Haskell-if.

3.2 Easy examples

With this operator in place, it is now possible to define continuous mathematical functions that contain branching. We will soon see that also other values than Spar-values will become useful in the branches. We can see that the `relu x` has sensitivity 1 since the first branch has sensitivity 1, the second branch has sensitivity 0 and both branches evaluate to zero at the decision boundary x=0. Similarly, `abs` implements the absolute value function, and we can observe that its sensitivity is 1 because both branches have sensitivity 1 and evaluate to zero at the decision boundary x=0.

```
relu :: Sen 1 Int Int
relu x = $(branch [| … |]
                  [|x > 0|]         -- condition
                  [|x|]             -- then
                  [|constant 0|] )  -- else
abs :: Sen 1 Int Int
abs x = $(branch  [| … |]
                  [|x > 0|]             -- condition
                  [|x|]                 -- then
                  [|constant 0 .- x|] ) -- else
```

3.3 Implementing cswp

The next program we want to implement is the cswp (compare and swap) function.

$$cswp(a,b) = \begin{cases} (a,b) & a > b \\ (b,a) & b \geq a \end{cases}$$

That is two-value sorting. Cswp takes a tuple and sorts it by placing the smaller elements as the first component of the tuple. This useful primitive can be used to implement Bubblesort/Insertionsort (by repeatedly comparing neighbors) and minimum and maximum functions. The later ones can be used in turn to implement Dijkstra's and the Bellman-Ford algorithms. This is why providing cswp as a primitive with assumed sensitivity 1 alleviates the pain of not allowing branching in related works ([1], [3]). We will implement cswp as a derived function using our branch operator. Doing so will allow us to implement the same more advanced algorithms. It will turn out that implementing cswp is not straight-forward, but we will need some iterations until we reach a satisfying implementation of cswp. On the way, we will learn more about the complexity of the branch operator.

```
cswp1 (x,y) = $(branch [|..|]
                                [|x > y|]   -- condition
                                [|(x,y)|]   -- then
                                [|(y,x)|] ) -- else
```

This is the direct translation for Spar using the new `branch` operator. Looking at the condition (x>y), we conclude the decision boundary $\{(x,y)|x = y\}$. Since the `branch` operator checks the agreement of both branches for every boundary value, the underlying SMT solver diverges — recall Section 2. Only a finite boundary can be fully obtained and checked (see Section 2.5.1 for more details).

We can get around this limitation with a small trick:[2] $x > y$ is equivalent to $x - y > 0$ and instead of calculating the difference in the condition, we can substitute it with variable $a = x - y$ and compare $a > 0$. The rewritten condition has the finite boundary $\{a|a = 0\}$. See the definition of `sswp2` in Fig. 3.3.

As a side effect, we do not have access to `x` and `y` inside the branches any longer. This is because the branch operator needs to analyze and evaluate the branches at compile time. For doing so, the branch-snippets are taken from its original context and executed and compared in plain environments (see Chapter 6 for more details). That means, that the original context is not available including surrounding functions and bound variables. Variables that take part in the condition (here `a`) will be instantiated when checking for all boundary values. But other variables (here `x` and `y`) are undefined in the compile-time environment, cannot be evaluated, and thus the checking fails if such variables are used. That means, our result must only depend on variable `a` denoting the difference.

It turns out, returning the pair $(a, 0)$ or $(0, a)$ otherwise works. These pairs are no

[2] That is similar how it is possible to define an order in a vector space by a positive cone.

```
sswp2 :: Dist n Int -> Dist n (Int, Int)
sswp2 a = $(branch [|…|]
                [| a > 0 |]
                [| (         a, constant 0) |]
                [| (constant 0,          a) |])

cswp2 :: Dist n (Int, Int) -> Dist (3 * n) (Int, Int)
cswp2 (x::Dist dx Int :*: y::Dist dy Int) =
    let (l :*: r) = sswp2 (x .- y) :: Dist n (Int, Int)
    in (l .+ y :*: r .+ y) :: Dist (n + 2*dy) (Int, Int)
```

Figure 3.3: Implement cswp by shifting by y.

more than $(x - y, 0)$ and $(0, x - y)$. When y is added on both sides, these pairs become (x, y) and (y, x).

As can be seen by the typing annotation in Figure 3.3, the sensitivity of `cswp2` is 3 rather than 1 as indicated by [3]. This comes from the overapproximation that Spar calculus is doing in tracking distances. When taking the subtraction `a`, the distances of x and y add up:

```
l :*: r = sswp2 (x .- y) :: Dist (dx + dy) Int.
```

We see in the line above that we subtract `y` with distance dy from `x` with distance dx. The value `x .- y` then has distance dx + dy and is given to `sswp2`. Since `sswp2` has sensitivity 1, the resulting tuple `l :*: r` will have the same distance type n = dx + dy. So, we can say that the distance type of the elements `l` and `r` add up to dx + dy, that is, dl + dr = dx + dy = n with dl denoting the distance of `l` and dr the distance of `r`.

Finally, y is added to both sides of the tuple, adding y's distance dy twice. The resulting distance can be further overestimated to $3n$, i.e. $dl + dy + dr + dy = n + 2dy \leq 3n$.

By naively applying ideas from [6] into Spar, we were able to define cswp for Spar, but with sensitivity 3 compared to sensitivity 1 as indicated by [3].

Rounding up

In this chapter we introduced the branch operator. It enables branching within the Spar-calculus. With this, it was able to define easy examples and the cswp function. Yet, it was not possible to infer sensitivity 1 for cswp. In the next Chapter, we will revisit cswp and define it with sensitivity 1.

4

Branching on refined types

As we have seen, `cswp2` has sensitivity 3. This is a very coarse and unpleasant overestimation. Considering what actually happens, the values of a tuple are maybe swapped, and thus the distances will not change. So the actual sensitivity is 1.

In this chapter we want to understand why this overestimation happens and improve our definition of cswp. This will be facilitated by specially designed types. Special care is needed not to break the sensitivity promise when doing so. At the end of this chapter, we will have constructed a sound cswp with sensitivity 1.

We start by checking why this overestimation occurs. Without loss of generality, let's consider (x<y) where `x :: Dist dx Int` and `y :: Dist dy Int`. We know that the result of `sswp2 (l :*: r)` will be `(constant 0, x .- y)`. After adding back y, the result is `(constant 0 .+ y, x .- y .+ y)`. The inferred type of the first part is `constant 0 .+ y :: Dist dy Int`, i.e. the inferred distance is `dy`. In contrast to this, `x .- y .+ y :: Dist (dx + dy + dy) Int` is typed with the distance `(dx + dy + dy)` because each `y` is treated independently and distances add up. However, from the context we know that both `y`s are exactly the same and thus the distances should cancel. So, the distance of `x .- y .+ y` should be effectively `dx`.

4.1 Liberal reversible differences

This leads to the notion of *reversible differences* of Spar-values. In the next paragraphs, we will introduce a deep embedding `Diff x y` of subtraction that generally behaves like the flat version `x .- y` and adds the components' distances. Additionally, it will provide the possibility to add back the subtrahend and cancel the connected distance, effectively yielding the minuend. This technique will help to typecheck `x .- y .+ y` with distance `dx`.

We introduce a new datatype `Diff` with accompanying functions on top of Spar to handle reversible differences in Figure 4.1. `Diff` wraps two Spar values. Intuitively, `Diff x y` denotes (x-y) and is fully characterized by the difference. We want to consider `Diff (constant 5) (constant 3)` and `Diff (constant 12) (constant 10)` to be equal since the differences are the same, i.e. 2. To actually evaluate the difference, `evaluateD` is provided. The function
`diffAddBack :: Diff dx dy -> Dist dy Int -> Diff dx Int` allows us to reverse a difference and add back the subtrahend: `diffAddBack (Diff x y) z = x`.

```
module Branching.ReversibleDiffs where

import NFuzz.API
import GHC.TypeLits

data Diff dx dy = Diff (Dist dx Int) (Dist dy Int)

cloneToNullSum :: Diff dx dy -> Diff dy dy
cloneToNullSum (Diff _ y) = Diff y y

evaluateD :: Diff dx dy -> Dist (dx + dy) Int
evaluateD (Diff x y) = x .- y

diffAddBack :: Diff dx dy -> Dist dy Int -> Dist dx Int
diffAddBack (Diff x y) z = x

data RTuple n dy =forall dx1 dx2. n~(dx1+dx2) =>
     RTuple (Diff dx1 dy) (Diff dx2 dy)
```

Figure 4.1: A simple implementation for reversible differences.

By reversing the difference and adding back the subtrahend, also the distances cancel. So this function alleviates the problem we observed before. Note, that `y` and `z` are not guaranteed to be the same value. For now, not passing the subtrahend z=y breaks the purpose of this function. In Section 4.4 we introduce a guarantee of only the subtrahend being a valid second argument for `diffAddBack`.

We still want to branch over the difference, so the variable in the condition (variable `a`) becomes a reversible difference. For giving runtime semantics to differences, we need a way to evaluate such values to `Int`s. This is provided by `evaluateD`.

We can now rewrite cswp using `Diff` as seen in Figure 4.2. Instead of passing the result of the subtraction to `sswp3`, we pass a richer reversible difference `a :: Dist dx dy`.

Intuitively, similarly to `sswp2`, `sswp3` returns either (a,0) or (0,a) which means that effectively the resulting pairs are (x - y, 0) or (0, x-y). That means that we are returning a pair with a difference.

Later, we want to add back the subtrahend `y` to the difference `Diff x y`. However, we do not know which side of the resulting pair will contain the difference to reverse. We solve this by making both sides a reversible difference adhering to the same subtrahend. So, we convert 0 into y - y. This is done by the primitive `cloneToNullSum`.

In `sswp2`, we have used `constant 0`. We replace it by a difference that evaluates to 0 while being compatible with subtrahend `y`. `cloneToNullSum` clones an existing

```
sswp3 :: Diff dx dy -> RTuple (dx + dy) dy
sswp3 a = $(branch [| … |]
                   [| a > 0 |]
                   [| RTuple a (cloneToNullSum a) |]
                   [| RTuple (cloneToNullSum a) a |])

cswp3 :: Dist n (Int, Int) -> Dist n (Int, Int)
cswp3 (x::Dist dx Int, y::Dist dy Int) =
    let (RTuple l r) = sswp3 (Diff x y) :: RTuple n dy
    in (diffAddBack l y:*: diffAddBack r y) :: Dist n (Int, Int)
```

Figure 4.2: Implement cswp using our `Diff` datatype.

difference `Diff x y` and creates a new one `Diff y y` based on the subtrahend, such that the result denotes 0.

Intuitively, the result of sswp3 is now either (x - y, y - y) or (y - y, x - y). Can we find a type that can carry both tuple values and the relevant type information? The answer is yes, and that type is `RTuple`.

We create a special type `RTuple` for tuples of `Diff`s with the same subtrahend. Semantically, `RTuple n dy` $\triangleq \{(x_1 - y, x_2 - y) | x_1 ::$ `Dist dx1 Int`$, x_2 ::$ `Dist dx2 Int`$, y ::$ `Dist dy Int`$, dx1 + dx2 = n\}$. Observe that both possible results from the branches can be captured by this type: $(x - y, y - y) \in$ `RTuple n dy` and $(y - y, x - y) \in$ `RTuple n dy`.

In Haskell, it is implemented as a pair of two `Diff`s. In the outer `cswp3`, the `RTuple` is split up and both contained reversible differences are reversed.

We want to keep track of the components' distances. Looking at the branches which shuffle the order of the components, the result could be of type `(Dist dx dy, Dist dy dy)` or `(Dist dy dy, Dist dx dy)`. So we cannot keep track of the exact minuends' distances, but their sum. `RTuple` is parameterized by two distance arguments. The first is the sum of the minuends' distances, the second stores the subtrahends' distances. Keeping track of the minuends' distances is enough to facilitate the later typechecking.

When reversing the differences in `cswp3` now, also the subtrahends' distances are reversed, and we obtain a cswp with sensitivity 1 only using the Spar API and our new operator.

4.2 Allowing more types in the condition

Let's take a closer look at the condition and reconsider the `abs` example. The function branches over `x>0`, where `x` is a Spar-value. In the condition, we compare some variable and integers.

```
abs x = $(branch  [| … |]
                  [|x > 0|]              -- condition
                  [|x|]                  -- then
                  [|constant 0 .- x|] )  -- else
```

Note that the condition does not make sense typewise. Variable `x` is of type `Dist`, while constant `0` is of type Int. This is no problem as the branch operator analyzes the condition syntactically, derives the boundary condition and gives the boundary-condition to the SMT solver. Semantically, we can think of Spar-values as annotated integers, so it makes sense. This intuition is backed by two functions for converting between Spar-values and integers. The function `constant` allows embedding an integer in Spar-values while the composite `(run . id)` extracts an integer. The extraction function is needed when substituting the branch operator invocation with an if-then-else for runtime. The carried integer is extracted so that it can be compared to the integer in the condition.

```
if ((run . id) x) > 0 then x
         else constant 0 .- x
```

For constructing the compile-time checks, it is the other way round (see Section 3.1). The variable `x` needs to be instantiated with the integers on the decision boundary (calculated by the SMT solver). So these integers need to be embedded as Spar-values. This finally explains the purpose of the first argument of the branch operator (as seen in Figure 4.2). A function that takes integers (those comming from the SMT solver) and returns values of the condition variable must be passed.

```
abs x = $(branch  [| \i -> constant i |]
                  [|x > 0|]              -- condition
                  [|x|]                  -- then
                  [|constant 0 .- x|] )  -- else
```

Note, how we inserted a function in the first argument. This function, wrapping Spar's `constant`, allows us to do the reverse at compile-time. From an integer, it allows constructing a Spar-value, that evaluates to the given integer.

Now, let us come back to the cswp example. In the previous section the `Diff` type was introduced to store richer structure of differences. It was shown that this richer structure is enough to be more precise about the subsequent calculations and finally show a sharper distance reversed differences. In this section, we will deal with the implications this brings, particularly using the `Diff` type in the condition.

As for Spar-values, we want to think about the differences as integers. That will allow us to continue using the SMT solver and write a condition like `a > 0`, while `a` is of type `Diff`. For this, again, embedding and extraction function are needed. In the previous section, it was already mentioned that `Diff` comes with the function `evaluateD` for extracting the integer out of a difference. Embedding an integer `x` works by thinking of `x` as `x - 0`. Then, `[| \x-> Diff (constant x) (constant 0) |]` embeds `x` in a `Diff` value. Similarly, the branch operator needs a way to translate integers to reversible differences in `sswp`. This is achieved by using `[| \i-> Diff (constant i) (constant 0) |]` as the first argument. At run-time, when applied to `evaluateD`,

```
1  sswp4 :: Diff da db -> RTuple (da + db) db
2  sswp4 a = $(branch [| … |]
3      [| a > 0 |]
4      [| RTuple (Diff (constant 17)(constant 17)) (cloneToNullSum a)|]
5      [| RTuple (cloneToNullSum a) a |])
6
7  cswp4 :: Dist n (Int, Int) -> Dist n (Int, Int)
8  cswp4 (x::Dist dx Int, y::Dist dy Int) =
9      let (RTuple l r) = sswp4 (Diff x y) :: RTuple n dy
10     in (diffAddBack l y:*: diffAddBack r y) :: Dist n (Int, Int)
```

Figure 4.3: Broken implementation of cswp using our simple `Diff` datatype.

the result becomes i-0.

```
sswp3 :: Diff dx dy -> RTuple (dx + dy) dy
sswp3 a = $(branch [| \i -> createDiff (constant i) (constant 0) |]
                [| a > 0 |]
                [| RTuple a (cloneToNullSum a) |]
                [| RTuple (cloneToNullSum a) a |])
```

This section showed that we need accompanying functions for conversion between type of variables occurring in the condition and integers. Since these functions depend on the type to branch on, they are expected to be provided by the user.

4.3 Breaking liberal reversible differences

In section 4.1, reversible differences were introduced to define cswp with sensitivity 1. Reversible differences are problematic though, they are too liberal. It turns out they are breaking the sensitivity promise given by the type system.

Consider Figure 4.3. The function is slightly changed: In line 4, instead of `a`, `Diff (constant 17) (constant 17)` is put into the `RTuple`. The program typechecks. At compile time when both branches are compared on the boundary value $a = 0$ with `a=Diff (constant 0) (constant 0)` being a difference that evaluates to zero, they appear equal: Intuitively,

$$\begin{aligned}
&\texttt{RTuple (Diff (constant 17) (constant 17)) (cloneToNullSum a)}\\
=&((17-17),(0-0))\\
=&(0,0)\\
=&((0-0),(0-0))\\
=&\texttt{RTuple (cloneToNullSum a) a}
\end{aligned}$$

That means, the branch operator does not reject the modified `sswp4`.

If we consider one execution with `x=constant 1` and `y=constant 0`, `cswp4 (x :*: y)` evaluates to (17,0).

```
  cswp4 (constant 1 :*: constant 0)
= let (RTuple l r) = RTuple (Diff 17 17) (Diff 0 0)
  in (diffAddBack l y:*: diffAddBack r y)
= (diffAddBack (Diff 17 17) y:*: diffAddBack (Diff 0 0) y)
= (17 :*: 0)
```

However, if we change `x` by one to `x=constant 0`, the output produced is (0,0). While we changed the input only by distance 1, the output jumped by a distance of 17. Clearly, we have broken our sensitivity promise given by the type-system. But where did we break it? The error lies in assuming the differences `Diff x1 y1` and `Diff x2 y2` to be equal as soon as (x1-y1)=(x2-y2) while both differences behave differently when applied to `diffAddBack _ z`. The property that equal values behave equally is called behavioral equality or extensional equality. We will come back to this in section 5.

4.4 Robust reversible differences

The underlying problem is that we want to keep track of connected values that are passed around separatedly: The difference `Diff x y` is connected to its subtrahend `y` which can be used to reverse it. However, the difference is passed through the branching while the subtrahend will be added back later.[1] We will introduce a type-level certificate that guarantees that the result of the branching is still connected to the subtrahend. This is achieved by introducing an existential type:

```
data Diff dx dy c = Diff (Dist dx Int) (Dist dy Int)
data Subtrahend dy c = Sub (Dist dy Int)
data ReversibleDiff = forall c. RDiff (Diff dx dy c, Subtrahend dy c)
```

Observe that we have extended the type `Diff` with an extra type-level phantom variable c. We added the type `Subtrahend` to capture subtrahends also with a phantom variable c. Finally, type `ReversibleDiff` connects both with an existential type. Each built `ReversibleDiff` will have a different existential type.

Figure 4.4 shows the full implementation of our extension to Spar. A pair of a difference and its corresponding subtrahend is obtained by destructing a `ReversibleDiff`. This introduces the existential type `c`. Type c is neither linked to any value, nor does it appear in the type of `ReversibleDiff`. Given some difference of type `Diff da db c1` and subtrahend of type `Subtrahend db c2`. Types `c1` and `c2` can only be unified (and thus used to reverse the difference), if the second argument of the difference matches the subtrahend. It is impossible to unify c with anything other than values stemming from the very same difference. Note that access to the type constructors `Diff` and `Sub` is restricted. Reversible differences are only available as `RDiff`-value by calling `createRDiff`. By this, `diffAddBack :: Diff dx dy c ->`

[1] The whole purpose of introducing the difference is to eliminate unbound variables in the branching.

```haskell
module Branching.ReversibleDiffs (
      --plain Diffs
      Diff()
      , evaluateD
      , createDiff
      , cloneToNullSum
      -- RDiffs
      , ReversibleDiff(RDiff)
      , createRDiff
      , Subtrahend()
      , diffAddBack
      -- Tuple
      , RTuple(..)
      ) where

import NFuzz.API
import GHC.TypeLits

-- plain Diffs
data Diff dx dy c = Diff (Dist dx Int) (Dist dy Int)

createDiff :: Dist dx Int -> Dist dy Int -> Diff dx dy ()
createDiff = Diff

cloneToNullSum :: Diff dx dy c -> Diff dy dy c
cloneToNullSum (Diff _ y) = Diff y y

evaluateD :: Diff dx dy c -> Dist (dx+dy) Int
evaluateD (Diff x y) = x .- y

-- reversible Diffs
data ReversibleDiff dx d =
      forall dy c. d~(dx+dy) => RDiff (Diff dx dy c, Subtrahend dy c)

newtype Subtrahend dy c = Sub (Dist dy Int)

createRDiff :: Dist dx Int -> Dist dy Int ->
      ReversibleDiff dx (dx+dy)
createRDiff x y = RDiff (Diff x y, Sub y)

diffAddBack :: Diff dx dy c -> Subtrahend dy c -> Dist dx Int
diffAddBack (Diff x _) (Sub _) = x

data RTuple n dy c =
      forall dx1 dx2. n~(dx1+dx2) =>
       RTuple (Diff dx1 dy c) (Diff dx2 dy c)
```


Figure 4.4: An implementation of reversible differences with certifications.

```
sswp5 :: Diff dx dy c -> RTuple (dx + dy) dy c
sswp5 a = $(branch [| \i -> createDiff (constant i) (constant 0) |]
                   [| a > 0 |]
                   [| RTuple a (cloneToNullSum a) |]
                   [| RTuple (cloneToNullSum a) a |])

cswp5 :: Dist n (Int, Int) -> Dist n (Int, Int)
cswp5 (x::Dist dx Int, y::Dist dy Int) =
    let
        RDiff (diff::Diff dx dy c) cy = createRDiff x y
        RTuple l r = sswp5 diff :: RTuple n dy c
    in (diffAddBack l cy:*: diffAddBack r cy) :: Dist n (Int, Int)
```

Figure 4.5: Implement cswp using certified differences.

`Sub dy c -> Dist dx Int` can only be applied to differences which agree on the subtrahend in b. Note that the code in Fig. 4.5 works purely on top of Spar, i.e. all guarantees given by Spar persist, particularly that the inferred distance of the result of `diffAddBack` is correct. Finally, a function `createDiff :: Dist dx Int -> Dist dy Int -> Diff dx dy ()` for creating reversible differences directly from two values is provided. Note that the resulting differences actually cannot be reversed, since it is impossible to construct a subtrahend with the right certificate `()`.

This function is useful for creating reversible differences on the decision boundary for compile-time checking. It will be used in the first argument of the branch operator and therefore for instantiating condition variables to values on the decision boundary when comparing both branches.

4.5 Revisiting cswp

Figure 4.5 shows the adapted implementation of cswp. The difference is now constructed using the function `createRDiff` and destructing the result. The certified difference (`diff`) is passed through the branching, keeping the certificate and yielding a pair of certified differences. The certified subtrahend (`cy`) is added back to the resulting pair of differences, which are certified to match the subtrahend.

Note that it is impossible to imitate our previous modification (returning `Diff 0 0`). We cannot create a new difference which adheres to the same certificate. This was the last step towards defining cswp with sensitivity 1 as a derived function using the branch operator.

5

Soundness

In the previous sections, we have created types for tracking reversible differences. It turned out that when designed without caution, they allow breaking sensitivity promises. In this chapter, we will show why our final revision does not break the sensitivity promises. In particular, we will show that our final implementation of cswp is sound.

For this, let's consider an input `p :: Dist n (Int, Int)` to cswp. This describes two executions with two values $p_1 \sim_n p_2$ with $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$. Looking at the types, we can introduce the subdistances $x_1 \sim_{dx} x_2$ and $y_1 \sim_{dy} y_2$ with $n = dx + dy$.

Let's track those values and distances through the program. We consider two different cases. First, the easy case is when both inputs are processed by the same branch. If $x_1 > y_1 \wedge x_2 > y_2$ or $x_1 \leq y_1 \wedge x_2 \leq y_2$, we could replace the branching by just the code of the respective branch without changing the values. Only `RTuple`s remain which do not break the Spar calculus. Thus, we can inherit the wished guarantee from the result being typed with `Dist n (Int, Int)`.

Second, consider, without loss of generality, $x_1 > y_1$ but $x_2 \leq y_2$. I.e. the results of both executions are based on different branches.

5.1 General proof idea

Let's consider a hypothetical third execution of cswp at point $p_3 = (x_3, y_3)$ which lies exactly on the decision boundary (i.e. $x_3 = y_3$) and exactly between p_1 and p_2. We can find p_3 by a sweep from p_1 to p_2 with side-condition $x = y$.

In our context, a sweep from some point a to point b is the direct connection between those two points: $\{a + \theta \cdot (b - a) | \theta \in [0, 1]\}$. It can be thought as smoothly moving from a to b by linear combinations of those points. The parameter θ describes how far we already traveled with $\theta = 0$ being at the start and $\theta = 1$ having reached the goal.

$$\{(x, y) = p_1 + \theta \cdot (p_2 - p_1) | \theta \in [0, 1], x = y\}$$

Let's search for θ_3 such that $p_3 = p_1 + \theta_3 \cdot (p_2 - p_1)$. We are looking for $p_3 = (x_3, y_3)$

with $x_3 = y_3$ (side-condition) and $x_3 = x_1 + \theta_3 \cdot (x_2 - x_1)$ and $y_3 = y_1 + \theta_3 \cdot (y_2 - y_1)$ (sweep-construction). So for p_3 it must hold $x_1 + \theta_3 \cdot (x_2 - x_1) = y_1 + \theta_3 \cdot (y_2 - y_1)$. We get

$$\theta_3 = \frac{x_1 - y_1}{y_2 - y_1 - x_2 + x_1} = \frac{\overbrace{x_1 - y_1}^{>0}}{\underbrace{y_2 - x_2}_{\geq 0} + \underbrace{x_1 - y_1}_{>0}}.$$

Since the denominator is non-zero, we conclude that this is the only possible θ_3. Further, $\theta_3 > 0$ since the nominator and denominator are strictly positive and $\theta_3 \leq 1$ since the nominator is smaller than or equals the denominator.[1]

We introduce the distances $m1, m2$ with $p_1 \sim_{m1} p_3 \sim_{m2} p_2$. It holds $n = m1 + m2$.

$$\begin{aligned} m1 + m2 &= |p_1 - p_3| + |p_3 - p_2| \\ &= |p_1 - (p_1 + \theta_3(p_2 - p_1))| + |p_1 + \theta_3(p_2 - p_1) - p_2| \\ &= |\theta_3(p_2 - p_1)| + |p_1 + \theta_3(p_2 - p_1) - p_2| \\ &= |\theta_3(p_2 - p_1)| + |(\theta_3 - 1)(p_2 - p_1)| \\ &= \theta_3|(p_2 - p_1)| + (1 - \theta_3)|(p_2 - p_1)| \\ &= |(p_2 - p_1)| = n \end{aligned}$$

Recall how cswp works: It takes a tuple, unpacks it, calculates the differences and gives this to `sswp`. There we branch over the difference and return a tuple of differences. Finally, the tuple is unpacked, the differences reversed, and the resulting minuends are put in a tuple and returned. For the sake of our analysis, we need to introduce some more notation. Let's think of the branches as two functions `b1, b2 :: Diff dx dy c -> RTuple (dx + dy) dy c`. Assume functions `pre :: Dist n (Int, Int) -> Diff dx dy c` and `post :: RTuple (dx + dy) dy c -> Dist n (Int, Int)` that contain all the processing around the branching. So `pre` does tuple-unpacking and creating the difference. `post` does `RTuple`-unpacking, difference reversing and tupling. The semantics of cswp now are expressible as `cswp5 = post . sswp5 . pre`.

We can now create functions
`f1 = post . b1 . pre :: Dist n (Int, Int) -> Dist n (Int, Int)` and
`f2 = post . b2 . pre :: Dist n (Int, Int) -> Dist n (Int, Int)`. They do not include branching but operate completely on the safe part of the Spar API. Let f_1, f_2 be the mathematical counterpart. The checked types tell us that both functions have sensitivity 1. So we can apply them to $p_1 \sim_{m1} p_3$ and the distance stays the same: $f_1(p_1) \sim_{m1} f_1(p_3)$. The same holds for f_2 and $p_3 \sim_{m2} p_2$. Remember, that we check that both branches have the same value at the decision boundary. I.e. we check that $f_1(p_3) = f_2(p_3)$.

[1] In general, we are dealing with two intersecting closed sets. First, the sweep between p_1 and p_2. Second, the set $S = \{(x, y) \mid x \leq y\}$. Now, $p_1 \notin S$ but $p_2 \in S$. It is possible to construct a converging sequence in the sweep and S that converges towards the boundary. We set p_3 as the convergence point.

Now we have all the pieces to make our analysis run!

$$\begin{aligned} \texttt{cswp}\ p_1 &\triangleq f_1(p_1) && (5.1)\\ &\sim_{m1} f_1(p_3) && (5.2)\\ &= f_2(p_3) && (5.3)\\ &\sim_{m2} f_2(p_2) && (5.4)\\ &\triangleq \texttt{cswp}\ p_2 && (5.5) \end{aligned}$$

Our argument works by using the compile-time check in line (5.3) for arguing that the value does not change, when going to the other branch. From this we follow $|\texttt{cswp}\ p_1 - \texttt{cswp}\ p_2| \leq m1 + m2 = n$.

5.2 Equality classes of same behavior

When checking at compile-time whether (5.3) holds for p_3, there is a problem. For every p_1, p_2 there is exactly one p_3, and we do not know the precise p_1, p_2 at compile-time. So we do not know which p_3 to check. The best hope is to check all possible p_3. But those are infinitely many (and not even all in $\mathbb{Z}^2$).

Remember, this is why we introduced the difference. We observed that the condition $x - y > 0$ only depends on the difference of the two values. This leads to equality classes of `Diff`-values with the same difference.

$$E_v = \{(x, y) \mid x - y = v\}$$

Now all values in a specific class E_v are processed by the same branch. And either all values of E_v lay on the boundary or none. Here, E_0 with difference 0 is the equality class on the boundary.

Looking at the implementation, we see that the branches cannot inspect inside a difference or construct new ones, but put it in one or the other side of a tuple. They cannot inspect a difference because they are lacking the right certified `Subtrahend` for doing so. The branches will behave the same for all elements of an equality class. That means, it is enough to check equality of both branches at one value for each equality class. We choose to check $p_3 = (0, 0)$. This handwavy explanation is made more precise and formal in Appendix A.

5.3 Compile-time checks

Now we want to see how the actual checks at compile time work. At compile-time, we construct a snippet of code that assesses this equality:

```
v1 = createDiff (constant 0) (constant 0)
print $ (b1 v1) == (b2 v1)
```

Here, the definition of `v1` is hard-coded for our example with $p_3 = (0, 0)$.

In general, the condition boundary could be different. If the condition was (x-y>1), we need to check $E_1 = \{(x, y)|x - y = 1\}$. We would choose $p_3 = (1, 0)$ and the snippet would become

```
v1 = createDiff (constant 1) (constant 0)
print $ (b1 v1) == (b2 v1)
```

If the condition was $(x - y > 1 \vee x - y < 3)$, the condition can flip when $x - y > 1$ flips or when $x - y < 3$ flips. So it has two boundaries. We need to check that both branches are the same for both boundaries, so two checks are performed. The boundaries are the equality classes E_1 and E_3. We create two snippets

```
v1 = createDiff (constant 1) (constant 0)
print $ (b1 v1) == (b2 v1)
```

and

```
v1 = createDiff (constant 3) (constant 0)
print $ (b1 v1) == (b2 v1)
```

Both snippets must result in `True` in order to accept the branching.

We can see a repeating pattern. The analysis of the condition yields equality classes which constitute the boundary. For every class, we construct a snippet with one representative of the class instantiated and perform equality check. For equality class E_v we chose to use $v, 0$ as the representative. In order to automate this at compile-time, we need a way to construct some value in an equality class E_v. We use the function `\v-> createDiff (constant v) (constant 0)` for this. For every equality class E_v it creates a representative. Since the types we branch over vary (for the easy examples, it was `Dist n Int`; here it is `Diff dx dy c`), this function cannot be hard-coded but is part of the branching interface.

This finally explains the first argument to the branch operator `[| \i-> createDiff (constant i) (constant 0) |]`. A function is expected for creating representatives on the decision boundary. It will be used in the checking snippets.

Rounding up

We have seen how `cswp` $p_1 \sim_n$ `cswp` p_2 and that cswp is sensitive with sensitivity 1. This works by taking a third point p_3 between p_1 and p_2 on the boundary. This point is part of an equality class E_0 whose elements all behave the same with respect to the branching. We statically check that the branches agree for $(0, 0) \in E_0$ and conclude that they also agree for p_3. Sensitivity of the branches is used for reasoning that the distance of the inputs is not increased.

In Appendix A, we will argue the same but introduce extensional equality in order to make the reasoning more crisp but also more complicated. In Appendix B, we will argue that this line of reasoning can be generalized. Four requirements are synthesized that, if met, guarantee soundness of branching.

6

Implementation in GHC

This chapter shows the details of the branch operator's current implementation. Our solution is written as a GHC plugin. It performs the aforementioned analysis and checks at compile time and injects a plain if-then-else in the final code. Currently, the plugin is written as a Template Haskell(TH) function which gets passed the arguments (condition, branches) as syntactic elements.

In the future, this may change to an implementation as source-plugin. This would allow having a prettier syntax.

The steps our plugin performs are:

1. Finding all branchings
2. Performing static analysis
3. Emiting code for runtime

In the following, those will be described in detail. This shows challenges faced during implementation and fills gaps in the description of the implementation given so far.

6.1 Finding all branchings

In this step, the branchings to be checked need to be identified. Using TH, this is straightforward, since the branch operator is explicitly called at every such branching. The branch operator is a function that runs at compile time. So our implementation behind the branch operator is directly called, and we do not need to implement a search for relevant branchings.

Note that plain Haskell branching (using if-then-else) does not work on Spar-values, so there is no danger of missing and not checking a relevant branching.

6.2 Static analysis

This is the most involved stage. Several substeps have to be performed:

1. Parsing the condition
2. Deriving the boundary condition

3. Calculating boundary values by searching values satisfying the boundary condition.
4. Call both branches on all boundary values and assert agreement

Parsing the condition The condition expression is expected to be written in plain Haskell. As the expression is available to the plugin as syntactic value, it is first parsed recursively to a `Condition` value. Currently the comparators `>=, >, ==` and logic combinators `not, &&, ||` are supported. Additionally, plain integers and bound variables are supported. Parsing is implemented in module `SparBranching.hs`.

The grammar of parseable conditions is

$$
\begin{aligned}
\mathit{Condition} &::= \mathit{BoolE} \\
\mathit{BoolE} &::= \mathit{BoolE}\ \texttt{\&\&}\ \mathit{BoolE} \mid \mathit{BoolE}\ \texttt{||}\ \mathit{BoolE} \mid \texttt{not}\ \mathit{BoolE} \mid \mathit{CMP} \\
\mathit{CMP} &::= \mathit{IntE}\texttt{==}\mathit{IntE} \mid \mathit{IntE}\texttt{>=}\mathit{IntE} \mid \mathit{IntE}\texttt{>}\mathit{IntE} \\
\mathit{IntE} &::= \text{Var}\ \mathit{STRING} \mid \text{Lit}\ \mathit{INTEGER} \mid \mathit{IntE}\texttt{+}\mathit{IntE} \mid \mathit{IntE}\texttt{-}\mathit{IntE}
\end{aligned}
$$

Deriving the boundary condition We transform the branch condition into another condition, the *boundary condition*. The boundary condition encodes when the branch condition flips. The transformation is implemented in the function `boundary`. It is stated in mathematical notation. As it works on the syntax, generating an expression from another expression, we use different font styles to distinguish variables (like e) from syntactic elements (like `==` or `&&`).

$$
\begin{aligned}
\text{boundary}(e1\texttt{==}e2) &= e1\texttt{==}e2 \\
\text{boundary}(e1\texttt{>}e2) &= e1\texttt{==}e2 \\
\text{boundary}(e1\texttt{>=}e2) &= e1\texttt{==}e2 \\
\text{boundary}(\texttt{not}\ e) &= \text{boundary}(e) \\
\text{boundary}(e1\texttt{\&\&}e2) &= e1\texttt{\&\&}\text{boundary}(e2)\ \texttt{||}\ e2\texttt{\&\&}\text{boundary}(e1)\ \texttt{||}\ \text{boundary}(e1)\texttt{\&\&}\text{boundary}(e2) \\
\text{boundary}(e1\texttt{||}e2) &= (\texttt{not}\ e1)\texttt{\&\&}\text{boundary}(e2)\ \texttt{||}\ (\texttt{not}\ e2)\texttt{\&\&}\text{boundary}(e1)\ \texttt{||} \\
&\qquad \text{boundary}(e1)\texttt{\&\&}\text{boundary}(e2)
\end{aligned}
$$

It works recursively on the syntax. The first cases are straightforward: A comparison flips when both sides are equal. A negated expression flips, when the original expression flips. For the conjunctions, let's consider the example expression `a > 0 || b == 0`. This condition flips if $b \neq 0$ and a becomes positive or if $a \leq 0$ and b becomes zero. In general, a conjunction flips if one side flips (recursively calculated by boundary) and the conjunction is not already determined by the other side. For the special case when both sides are identical, we need to include boundary($e1$)`&&`boundary($e2$).

Calculating the boundary values We employ an SMT solver for finding all boundary assignments. For this, we use the SimpleSMT package (see Section 2.4). First, we register all variables and the boundary condition in the solver. Then, we obtain satisfying value assignments, i.e. values on the boundary, one by one (see Section 2.5.2).

Compare branches In this step, we evaluate both branches for every boundary value assignment and assert agreement. Let `$(branch` *lit c b1 b2*) be a branching to be checked and [`v1` = *w1*; . . . ; `vn` = *wn*] a variable assignment on the condition boundary. Again, we mix variables containing syntactical objects with direct code. To distinguish this, we use different font styles for `plain code` and *variables*. The variables contain syntactical pieces that are copied at the respective location. We construct a temporary Haskell snippet, that will be evaluated.

```
let fromInt = lit
    v1 = fromInt w1   }
    ...               } variables in the condition
    vn = fromInt wn   }
    resultB1 = b1 ← code for then-branch
    resultB2 = b2 ← code for else-branch
 in resultB1 eq0 resultB2
```

First, we insert the embedding of integers into the condition variables types. Then we build a local context by assigning the variables present in the condition. Finally, both branches are inserted and their result compared using the `eq0` function. We expect the branches to only depend on condition variables, so they are closed in this temporary string.

Let's look at this easy example to make this more explicit.

```
abs x = $(branch  [| \i -> constant i |]
                  [|x > 0|]              -- condition
                  [|x|]                  -- then
                  [|constant 0 .- x|] )  -- else
```

The boundary condition is $x = 0$ with the single fulfilling value $x = 0$. Thus, we will create one snippet for this value. The only variable occuring in the condition is `x`. This is the generated snippet:

```
let fromInt = \i -> constant i
    x = fromInt~0
    resultB1 = x
    resultB2 = constant 0 .- x
in resultB1 eq0 resultB2
```

This string construction and evaluation is done for every boundary value assignment. If both branches agree on all values assignments, the analysis stage of the plugin is passed.

6.3 Emitting run-time code

Let again `$(branch` *lit c b1 b2*) be a branching. Then this translates to

```
if c' then
    b1
else
    b2.
```

Notice, that we use `c'` instead of `c`. The problem is that c is syntactically comparing condition variables to integers. In the easy examples condition variables are Sparvalues, for cswp the variables are of type `Diff`. Comparing these or other non-integer types to integers does not work. The branch operator expects an instance of the typeclass `ToInt t` with corresponding function `toInt::t -> Int` to be implemented for types used in the condition variables.

Condition `c'` is then obtained by syntactically injecting a call to `toInt` in front of every variable in the condition `c`. After checking it, the branch operator turns the `abs` function into

```
abs x = if (toInt x) > 0  then
            x
        else
            constant 0 .- x
```

RoundingUp

This chapter showed the current implementation details of our branch operator. We focused on presenting the challenges and our current solution to them.

Using TH is not the only way the branch operator could be implemented. TH was chosen because it makes Step 1 easy and has good support for syntactic analyses and modifications of code and is sufficiently well-documented. The plugin API of GHC is not stable yet and changes even within minor versions, so the up-to-date documentation and existence of introductions was a big advantage of TH.

In the future, the branch operator might be implemented as a Code Plugin. This allows a more flexible, and thus prettier, syntax, but makes Step 1 possibly harder. We expect the challenges of Steps 2 and 3 to be mostly independent of the concrete plugin technology.

7

Advanced examples

In this chapter, we will look at more examples. As described in Section 2.3, earlier frameworks for sensitive functions, namely Fuzz[3], have chosen to provide cswp as a primitive and derive more elaborate functions from it. Now that we have cswp in place, we can pick the same fruits. Cswp makes it easy to implement derived functions like min and max but also primitive sorting algorithms like Bubblesort. In the following we want to look beyond what can be derived with cswp and implement Mergesort.

7.1 Mergesort

We implement Mergesort using our branch operator. MergeSort is a sorting algorithm that works by splitting up the list to be sorted, recursively sorting both parts and merging them again. While merging, the heads of the sorted parts have to be compared iteratively and the smaller (w.l.o.g.) element is removed from its part and appended at the result. This is repeated until one of the parts is empty and the remaining other part is appended.

The difficult step when dealing with Spar-values here is merging: it involves comparing Spar-values and removing an element from one of the parts. The Spar-API does not provide comparison of Spar-values and changing the remaining parts brings the danger of leaking information if their lengths can be observed. To solve this problem we will introduce specifically crafted data types. For those, we establish invariants that hold during the process, finally yielding the correct result.

7.1.1 Implementation

We implement Mergesort in two steps: First, the *main sorting procedure* splits the input vector in two parts, recursively calls Mergesort on those parts and merges the sorted results of the recursive calls. It takes a decision function as an argument that is called while merging for deciding which tip to move from its sorted part to the final result. This decision function encodes basically a comparison. The main procedure does not use the branch operator, it just needs to apply the decision function.

Secondly, we implement a *decision function* encoding the normal smaller-or-equal on natural numbers. This function needs to branch.

By splitting up Mergesort like this, we ease the reasoning: We will establish properties on the main procedure which do not depend on branching. Then, we will use these properties to reason about the decision function.

We start top-bottom with the main sorting procedure.

Main Sorting Procedure

```
data ListPair d l = forall l1 l2. (l~(l1+l2)) =>
  ListPair (Dist d (Vec l1 Int, Vec l2 Int))

mergeSort' :: Dist d (Vec l Int) -> DecFun -> Dist d (Vec l Int)
mergeSort' v f | lengthV v < 2 = v
mergeSort' v f                 = mergeSort'' $ splitV v
    where
        mergeSort'' :: ListPair d l -> Dist d (Vec l Int)
        mergeSort'' (ListPair (v1 :*: v2)) =
          merge (mergeSort' v1 f) (mergeSort' v2 f) f

splitV :: Dist d (Vec l Int) -> ListPair d l
splitV v = headTail (div (lengthV v) 2) (Nil::Dist 0 (Vec 0 Int)) v

headTail :: Int -> Dist d1 (Vec l1 Int) -> Dist d2 (Vec l2 Int)
  -> ListPair (d1+d2) (l1+l2)
headTail n acc is | n == 0 = ListPair $ reverseV acc :*: is
headTail n acc (i :> is)   = headTail (n - 1) (i :> acc) is
```

Abstract sorting The datatype `ListPair` stores a pair of lists and only keeps track of the sums of distances and lengths. `mergeSort'` is the main sorting procedure. Given a list and a decision function (we will discuss the type in the paragraph **The Decision Function** on page 41), it splits the list, sorts both parts and merges those. The functions `splitV` and `headTail` are helper functions for splitting a list.

Merging Next, we consider merging. Merging is facilitated by the datatype `PartialMerge`.

```
data PartialMerge d l =
  forall d1 d2 l1 l2 l3 d3. (d~(d1+d2+d3), l~(l1+l2+l3))
  => PartialMerge (Dist d1 (Vec l1 Int))
                  (Dist d2 (Vec l2 Int))
                  (Dist d3 (Vec l3 Int))

startPartialMerge :: Dist d1 (Vec l1 Int)
  -> Dist d2 (Vec l2 Int) -> PartialMerge (d1 + d2) (l1 + l2)
startPartialMerge = PartialMerge (Nil::Dist 0 (Vec 0 Int))

isMerged :: PartialMerge d l -> Bool
isMerged (PartialMerge _ Nil Nil) = True
isMerged PartialMerge {} = False

getResult :: PartialMerge d l -> Dist d (Vec l Int)
getResult  (PartialMerge resultAcc Nil Nil) = reverseV resultAcc
```

It encodes three lists and only keeps track of the summed lengths and distances. The first list stores the accumulated result that is already merged. The second and third list store the remaining parts that still need to be merged. Merging starts by constructing a `PartialMerge` with empty first list and storing the lists to be merged in the latter lists. Then this `PartialMerge` is iteratively progressed, i.e. one element of one of the latter lists is moved into the result accumulator.

```
progressFromLeft :: PartialMerge d l -> PartialMerge d l
progressFromLeft (PartialMerge resultAcc (l:>ls) rs) =
    PartialMerge (l :> resultAcc) ls rs

progressFromRight :: PartialMerge d l -> PartialMerge d l
progressFromRight (PartialMerge resultAcc ls (r:>rs)) =
    PartialMerge (r :> resultAcc) ls rs

progressWithComparison :: PartialMerge d l -> DecFun -> PartialMerge d l
```

There are three functions for progressing. `progressFromLeft` and `progressFromRight` move an element from the respective side to the result accumulator. The interesting progression is `progressWithComparison`. Given a `PartialMerge` and a decision function (the comparison), it uses the decision function to decide which part's tip should be moved to the accumulated result.

With this, merging becomes creating an initial `PartialMerge` and progressing it until both remaining parts are empty and return the result. Note that the inputs to `merge` are assumed to be already sorted with respect to the same decision function.

```
merge :: Dist d1 (Vec l1 Int) -> Dist d2 (Vec l2 Int)
  -> DecFun -> Dist (d1 + d2) (Vec (l1 + l2) Int)
merge v1 v2 f = merge' (startPartialMerge v1 v2)
    where
        merge' pm  | isMerged pm = getResult pm
        merge' pm = merge' (progressWithComparison pm f)
```

Making progress with a decision function In the previous paragraph, we have seen that `merge` progresses the `PartialMerge` iteratively with the help of a decision function. Such a function is given the difference[1] between the two tips of the remaining parts, i.e. the candidates for moving over to the result, and expected to choose one candidate.

For instance, if the sorted lists are $[x1, x2, \ldots]$ and $[y1, y2, \ldots]$ with $x1 > y1$, the difference $x1 - y1$ is given to the decision function. Let's say the decision encodes the smaller-than relation, then it gets a positive number as input and indicates to go left.

The following types constitute the interface between the main sorting procedure and the decision function.

```
type DecFun = forall c. CDist c -> MergeDec c
data CDist c = forall d. CDist (Dist d Int)
data MergeDec c = L (CDist c) | R (CDist c)
```

The `CDist` wraps a Spar-value containing the difference between the candidates. The decision function then returns either `L` or `R`. Note that a decision (`L` or `R`) carries a certified value as payload. Given that the `CDist` constructor is abstract and the `DecFun` type requires the certificate types `c` to match, the payload will be the difference the decision was based on. For the further processing of the decision, only the constructors are significant, the payload is important for the compile-time checks and will be discussed later.

An example decision function could look like this:

```
df c@(CDist diff) = if run id diff > 0 then L c else R c
```

Note, that this runs the Spar-value `diff` and this considered is unsafe. We will iterate this decision function in the next section.

[1] The difference between the candidates is enough to determine which side to progress with. Taking the difference is an idea shared with the cswp example.

```
progressWithComparison :: PartialMerge d l -> DecFun -> PartialMerge d l
progressWithComparison pm@(PartialMerge _ (l:>ls) Nil) f =
    progressFromLeft pm
progressWithComparison pm@(PartialMerge _ Nil (r:>rs)) f =
    progressFromRight pm
progressWithComparison pm@(PartialMerge _ (l:>ls) (r:>rs)) decFun =
    case decFun $ CDist (l .- r) of
        L _ -> progressFromLeft pm
        R _ -> progressFromRight pm
```

A `PartialMerge` can be progressed with a decision function. If one of the two parts is empty, there is nothing to decide. It is clear that an element of the other part has to be moved over. If both remaining parts have elements left, the decision function is called on the difference of the candidates. Then, the `PartialMerge` is progressed according to the decision.

With this, we have seen all the parts of our Mergesort implementation for Spar-values. On the outermost layer, we split a list, sort it recursively and merge the two sorted parts. Merging creates a `PartialMerge` with empty result accumulator that is filled step by step. This is done by iteratively progressing the `PartialMerge` with the help of the decision function.

The decision function

We employ a decision function that implements greater-than on natural numbers. `branchCDist`, `goL` and `goR` are exported wrappers for the constructors `CDist`, `L` and `R`, respectively. In order to branch on the difference, a Spar-value, our branch operator is used. If the difference is greater than zero, the function decides to proceed with the left part, otherwise with the right one. Finally, `mergeSort` is implemented by using this decision function in combination with the abstract main sorting procedure.

```
df :: DecFun
df diff = $(branch [| \i -> (branchCDist . constant) i |]
            [| diff > 0|]
            [| goL diff |]
            [| goR diff |]
           )
mergeSort :: Dist d (Vec l Int) -> Dist d (Vec l Int)
mergeSort v = mergeSort' v df
```

7.1.2 Correctness

The implementation aligns with the standard definition of Mergesort. The input is split, recursively sorted and merged. Merging works by iteratively comparing the tips of the two remaining parts and moving the bigger element over. In our case, finding the bigger element is outsourced in the decision function. This function is easy to analyse and implements the well-known order on natural numbers.

7.1.3 Soundness

In this section, we argue that our implementation and branching do not break the sensitivity promise. This does not depend on the particular implementation of the decision function, but is solely guaranteed by the compile-time checks of the branch operator.

The abstract main sorting procedure does not need to branch and is written on top of Spar. The only code where we branch and may break the sensitivity promise is the decision function. Branching is done using our branch operator and specifically crafted types. We will argue that the use of the branch operator in combination with these types cannot break the sensitivity either.

The central datatype for merging is `PartialMerge`. For all the instances occurring during merging, some invariants hold.

- **Two lists of remaining elements are sorted with respect to the decision function.** `merge` is hidden and only called by `mergeSort`. By induction on the input length, it can be concluded that recursively sorting works and merging is only started on sorted inputs. So invariant holds for the initial `PartialMerge`. While progressing, elements are only taken from the remaining parts, never added or moved around. The sorting is not altered.
- **All elements in the result accumulator are greater or equal than the elements in the remaining lists.** This is true for the initial `PartialMerge`. The two tips are largest elements in the respective remaining parts. In every progression step, the bigger one of the two tips is moved to the result accumulator. So the element moved over is bigger or equal than every element in the part it comes from. It is also bigger or equal than the other tip and thus all the elements in the other part. Moving this element over keeps the invariant valid.
- **The result accumulator is sorted inversely.** This is true for the initial `PartialMerge`. Each progression, the element moved over comes from a remaining part. With the previous invariant, this implies that this element is smaller or equal than all the elements in the result accumulator. Moving that element to the tip of the result accumulator keeps its inverse sorting intact.

The dangerous situation in which sensitivity might be broken is when the branching or the decision function return a different constructor (`L` or `R`) for a small change in the input. The return value could be matched on and result in completely different later computations.[2]

Different constructors only possible for equal tips.

Using our branch operator branching requires both branches to be considered equal at the decision boundary. That is where the payload of the decision comes into play. Equality on `MergeDec` is defined by

[2] If the constructor stays the same, a pattern matching on it would match the same case and the only difference visible is the sensitive payload. This cannot break the sensitivity promise.

```
instance BranchResult (MergeDec c1) (MergeDec c2) where
  eq (L diff1) (R diff2) = toInt diff1 == 0 &&  toInt diff2 == 0
  eq (R diff1) (L diff2) = toInt diff1 == 0 &&  toInt diff2 == 0
  eq _ _ = True
```

So either both branches use the same constructor (not breaking the sensitivity promise). Or the branches use different constructors, but then the payload must equal zero. We link the payload of the result to the difference passed to the decision function. By this, the branch operator enforces that the branches can only use different constructors, if, when the condition flips, the difference is zero, i.e. both tips are equal.

Linking the value in the result payload to the difference passed works similar to the cswp example. The value to branch on is wrapped in a type with limited accessibility. Here, only the numerical value can be extracted. Particularly the constructor is hidden, preventing the branches to construct instances of this type. Yet, the result type `MergeDec` requires such an instance as payload. The only instances available are the ones passed to the decision function. In order to prevent reuse of instances, the passed differences and result values are annotated by a type variable c (certificate). The type of decision function requires that the function works for all certificates, effectively enforcing that the payload of the result was the value passed as input.

Merging yields a series of equal `PartialMerge`s.

First, we need to introduce, what equality on `PartialMerge` means.

Definition 4. *Two `PartialMerge` instances are considered equal if their result accumulators equal and their remaining parts contain the same elements. I.e. the remaining parts of an instance are thought of as a single bag and then the bags of both instances must equal.*

Given a `PartialMerge` with equal tips of the remaining parts, this allows exactly considering the results of both possible progressions equal. For fully merged `PartialMerge` instances, equality implies equality of the merge results.

We will now show that progressing `PartialMerge`s preserves equality and thus by induction equal `PartialMerge`s result in equal merge results. Let's consider the function `progressWithComparison` with two equal inputs pm_1 and pm_2. By being equal, it is clear that the smallest elements of their remaining parts equal. As we have seen in the invariants, the remaining parts are sorted, so it is clear that a smallest element is at a tip. We can think of the decision function as a comparison (see Appendix C for a more involved discussion) and for each `PartialMerge`, it points to a smallest element of the remaining parts. Moving this smallest element over, which must be equal for both pm_1 and pm_2, also keeps the results equal.

This observation can be iterated (induction over the count of element in the remaining parts) until the remaining parts are empty. Then the accumulated result and thus the merge result equal. I.e. progressing equal `PartialMerge`s yield equal final merge results.

In particular, given a `PartialMerge` with two equal tips, the progressed `PartialMerge` instances will equal, no matter which constructor was returned by the decision function. Progressing those equal `PartialMerge`s further will result in equal merge results. Equal instances of `MergeDec` behave equally, even if constructed differently.

Sensitivity preserved

In the following paragraph we use the fact that equal `MergeDec` instances result in equal merge results for arguing that `merge` is sensitive.

Consider two pairs of parts to be merged $(p11, p12) \sim_m (p21, p22)$ with $p11 \sim_{m1} p21$. We think of two executions `merge p11 p12` and `merge p21 p22`. Similarly to our reasoning for cswp, we now consider a smooth transition from one pair into the other and observe what happens at the decision boundaries. Without loss of generality, we only consider $p11$ morphing into $p21$. We define $px1 = p11 + \theta \cdot (p21 - p11)$ and observe `merge px1 p12`. Particularly, we observe the results (i.e. `L` or `R`) of the decision function calls. The merging process consists of calculating differences d_i between the tips, calling the decision function which responds with r_i`::MergeDec` and progressing.

If there is no change in those results r_i, `progressWithComparison` progresses with the same side and there is no change in behavior of the function. Note that we are only interested in the constructor returned by the decision function here, as this is what the constitutes the behavior. The remaining steps and functions are fully on top of Spar and the sensitivity promise is not broken.

Consider a change of a decision r_k of a decision function call around a specific θ', yielding values on the decision boundary in the branch. Let d_k denote the difference the decision r_k was based on. As shown in earlier paragraphs, d_k must be zero due to compile-time checks. So at progression k while merging, two equal tips are compared. We have seen that the result of a progression stays equal, when different constructors are used but the tips equal and, that the change in the intermediate decision does not change the merge result. So the merge result around θ' is stable.

Sensitivity can now be concluded piecewise. We start with with $\theta = 0$ to $\theta = \theta_1$, the last θ before a decision function result changes, and define $\texttt{px1}_1 = p11 + \theta_1(p21 - p11)$ and $\texttt{px1}_1 \sim_{n1} p11$. `merge p11 p12` $\sim_{n1}$ `merge px1_1 p12` since we did not branch. Progressing θ minimally past θ_1, so that some decision function result changed (θ_2, $\texttt{px1}_2$), we have seen that the merge result stays the same: `merge p11 p12` $\sim_{n1}$ `merge px1_1 p12` = `merge px1_2 p12`. Let θ_3 be the second occasion (if any) that the decision function result changes. Between θ_2 and θ_3 with $p11 + \theta_3 \cdot (p22 - p12) = \texttt{px1}_3 \sim_{n2} \texttt{px1}_2$, we can again use the argument, that merging effectively does not branch, and thus the sensitivity cannot be broken.

Taking these two sections together, we can reason that

$$\texttt{merge p11 p12} \tag{7.1}$$
$$\sim_{n1} \texttt{merge px1\_1 p12} \tag{7.2}$$
$$= \texttt{merge px1\_2 p12} \tag{7.3}$$
$$\sim_{n2} \texttt{merge px1\_3 p12}. \tag{7.4}$$

I.e. `merge` is sensitive on these two sections with respect to the first parameter. This construction can be iterated. A section without branching does not break the sensitivity promise and crossing the decision boundary does not change the result. Finally, it can be shown that `merge p11 p12` $\sim_{m1}$ `merge p21 p12`.

Analogously, `p12` can be morphed into `p22` while sensitivity is preserved. In the end, we have morphed `(p11, p12)` into `(p21, p22)` and observed that `merge`'s sensitivity is bounded by 1 during the whole process. Sensitivity of Mergesort follows by typing.

Quirks

There are some small details in our construction we have not yet considered while being important for not breaking sensitivity.

Given `PartialMerge` instances with equal tips of the remaining parts, we have seen that the decision function's result does not matter for the final merge result. However, it changes the shape of the remaining parts. `PartialMerge` is abstract, so that the lengths of the remaining parts cannot be observed outside `progressWithComparison`. Additionally, the amount of comparisons needed changes depending on the actual decisions. In order not to break sensitivity, we made sure that the amount of progressions needed does not depend on the amount of comparisons needed. In that way, again, no difference is observable from `merge`.

Another thing that changes depending on the actual decision is the actual tips compared and thus the differences given to the decision function. By making the decision function a pure function that is passed to `progressWithComparison`, it can observe the different values but cannot leak this information.

`merge` needs to be called on sorted lists. It is thus not exported but only accessible from `mergeSort`.

Rounding up

In this section, we have seen derived applications of the branch operator with the cswp function as intermediate step. One of the examples is the slow Bubblesort. We have implemented MergeSort as a faster sorting algorithm. Its speedup originates from making use of the knowledge whether cswp would need to flip the values or not. This is sensitive information and not available using the Spar-API or cswp. Hence for implementing Mergesort, we could not reuse cswp. Instead, we have defined Mergesort as an abstract sorting algorithm depending on a *decision function*.

8

Related Work

Frameworks for Differential Privacy In 2009 McSherry[2] presented an abstract framework for implementing DP. This gives analysts a set of predefined queries (with sensitivities annotated) to run against the database and which they can freely combine. From there, combining Differential Privacy with programming languages techniques has been an active field of research. Barthe, Gaboardi, Hsu, *et al.*[11] provide an overview about the different approaches. In 2021, DPella Lobo-Vesga, Russo, and Gaboardi[12] implemented a DP framework in Haskell. An additional innovation which came with DPella is that accuracies of queries can be estimated statically (without running them).

DP Frameworks for sensitivity In 2010 Reed and Pierce[3] presented Fuzz, a more relaxed calculus for DP. Queries can freely be defined from small building blocks with known sensitivity. A strong type system based on linear types is used to keep track of a query's sensitivity. It works by annotating how sensitive the output is with respect to the input. Further, a monad is provided to define executions and combinations of such queries.

Building on Fuzz, later works Near, Darais, Abuah, *et al.*[13] and Winograd-Cort, Haeberlen, Roth, *et al.*[14] followed in the tradition of using linear types.

Gaboardi, Haeberlen, Hsu, *et al.*[4] presented with dFuzz an implementation of Fuzz based on dependent types.

Abuah, Darais, and Near[5] presented the framework Solo for defining sensitive queries and running them in Haskell. This is a huge improvement as this means that this framework can be implemented in mainstream programming languages with a decent type system. Looking at the published sources, this framework is partially implemented and works by attaching sensitivities to globally defined data sources instead of functions.

Lobo-Vesga[1], [15] recently presented Spar, another framework for defining sensitive queries and tracking the sensitivities in Haskell's type system. Again, this is achieved by moving the type annotations from functions to values. Here, sensitivity is interpreted as an upper bound on how a function scales the distance between two inputs. As we also have seen in this work, this can be combined with polymorphism to express sensitivity.

Other sensitivity analyses The work by Chaudhuri, Gulwani, and Lublinerman[6] proposes a calculus to reason about robustness, i.e. sensitivity, of imperative programs. Syntactic rules are given for keeping track of dependencies of inputs of code snippets to outputs. Those are stored in Lipschitz matrices. Although they are not dealing with Differential Privacy directly, their insights on sensitivity are very interesting. Of particular interest to us is that they allow branching in their calculus. The derivation rules allow nested branchings but do not add the insight of taking a branch to the nested context.

Barthe, Eilers, Georgiou, *et al.* develop in [16] a calculus to reason about relational properties of imperative programs. This calculus constructs an expression in trace logic bottom up, so it results in a single global expression to be checked by a SAT solver. This implies, that context can be taken into account when considering nested branching. As sensitivity can be expressed as a relational property, we could get inspiration from this calculus. Constructing one global expression allows taking nesting of conditionals into account whereas our current approach only allows to reason on local scale and ignoring context knowledge. Using a solver for checking sensitivity comes with the cost that all values must be expressible for the SMT solver and the extra effort of arguing the equivalence of Haskell computations and SMT reasoning. This is an idea that could be explored in the future.

Jha and Raskhodnikova follow in [17] a different path and estimate Lipschitz continuity by black-box testing. This is connected to Differential Privacy fruitfully as this skips all the hassle of limiting expressiveness in order to track sensitivity.

Branching and examples DP frameworks (Fuzz[3], dFuzz[4], Solo[5]) do not provide arbitrary sensitive branching. In cases where branching is of interest, the introduction of the special operation *cswp* with hardcoded sensitivity was enough to implement those. Algorithms which needed more expressive branching are not impossible to implement.

The calculi for reasoning about sensitivity presented in [6] and [16] allow branching. Chaudhuri, Gulwani, and Lublinerman in [6] hints explicitly for Mergesort to be a feasible algorithm for their analysis.

Both works consist of deriving formulas fed to and checked by external SMT solvers. In contrast to this, our branch operator works (mostly) in pure Haskell. This brings limitations in reasoning (no inifinite decision boundary, no context knowledge) and flexibility in values (arbitrary Haskell values allowed in branches).

9

Conclusion

In the context of Differential Privacy, understanding the sensitivity of a query is essential. Spar is a DSL embedded in Haskell for writing such queries that translates the problem of understanding a query's sensitivity into a type-checking problem such that type inference becomes bounding the sensitivity. This DSL is of limited expressiveness, in particular no branching is possible.

What we did We have defined a branch operator for Spar in Template Haskell that allows branchings and injects compile-time checks to ensure that the branchings do not break sensitivity.

The branch operator analyses a branching condition, computes the decision boundary and asserts that both branches agree on this boundary. This is done by creating temporary Haskell snippets that are run at compile-time.

From there, we were able to reproduce sensitive algorithms that traditionally need branching without introducing a special purpose extra primitive cswp. Namely minimum, maximum and the absolute value function as well BubbleSort.

Additionally, we implemented MergeSort as a sensitive algorithm that heavily relies on branching but cannot be expressed with cswp.

Common patterns in cswp and mergesort In this thesis two examples were implemented: cswp and MergeSort. Both required specifically crafted types to support the reasoning of soundness. Specifically, both types used the introduction of a type level certificate.

For cswp, it was necessary to only allow differences `Diff x y` to be reversed with the right subtrahend `Sub y`.

For MergeSort, we needed to make sure that the payload of the decision matches the input.

Introducing certificates together with polymorphism over certificates and limited APIs can be thought of as a general pattern for controlling the possible interaction with type instances in different scopes. Specifically, speaking of cswp again, the certificate becomes a way to restrict access to the difference's components within the branch and to make sure that the result matches the right subtrahend rather

than a way to make sure that the right value is added mathematically. In fact, the subtrahend provided to `reverse` is not used.

How hard to apply? How useful? The final branch operator is rather flexible and allowed us to implement different branching algorithms. Employing the branch operator requires the developer to reason about equality on the types used as variables in the condition as well as about equality on the resulting types. For Spar-values, this is straightforward but becomes more involved for Robust Reversible Differences (used for cswp) and Merge Decisions (used for Mergesort). This has to be done once per type and can be used in different branchings on the same types. Hence, branchings on types that were used in this thesis (besides Spar-values) can reuse the reasoning presented and are thus easy to use. Unfortunately, the need for finite decision boundaries quickly makes it necessary to branch on specifically created types.

Was this worth the effort? This work researched one approach to allow general branching with sensitivity checking. Such branching is necessary for interesting and fast general differentially private algorithms. We have found fundamental difficulties and explored our approach for solving them.

For checking the agreement on the decision boundary, we decided to analyse the condition syntactically, use an SMT solver to enumerate the boundary and create temporary Haskell snippets for checking equality.

Enumerating the boundary only works for finite boundaries. Branching on differences was used to circumvent infinite boundaries. This made it necessary to introduce specific types, establish equality classes of same behavior on them.

One result the introduction of a typelevel certificate for controlling interaction with values in different parts of the code. This was facilitated by the introduction of type level certificates.

What alternatives would have been there? The SMT solver can be applied at different levels. We used it "only" for understanding the decision boundary.

Going one step further, I could also be used for checking agreement on the boundary. For doing so, the branches' semantics need to be translated into something the SMT solver can reason about. As a result, one formula would be constructed for each branching and fed to the SMT solver. If it is considered true, that would imply sensitivity.

It is also possible to use the SMT solver to reason about the sensitivity of function directly. Sensitivity can be expressed as a relational property. Doing so and following [16], a global formula could be constructed that is sufficient for sensitivity of a function. This formula could then be checked by an SMT solver. This makes it necessary to translate all Haskell semantics to the SMT solver.

9.1 Comparison to other solutions — Discussion

9.1.1 Advantages

Our approach is the path of the least divergence from Haskell. The branch operator is just a module that can be loaded and provides compile-time checking for branching. It works as a GHC plugin.

This implies that Haskell can be used freely in the branches. Particularly, the flexible types and integration allows using recursion.

9.1.2 Limitations

As we have seen, the approach of checking agreement on every value on the boundary requires a finite boundary.

Often, the possible values a variable can carry is limited if the context is taken into account. For example if a branching occurs in a branch of an outer branching. Taking this context knowledge into account, it is sometimes possible to reason sensitivity of branching in this particular context, that might not be sensitive in another context. Our operator does not take this context into account and will thus discard such branchings.

As the branches are evaluated at compile-time, the variables occurring need to be instanstiated. But as the branches are taken out of context and compared at compile-time, variables that do not occur in the condition are undefined and cannot be used in the branches.

Introducing specifically crafted types is sometimes necessary and comes with the extra work of reasoning about the extensionality of equality on these types.

As a limitation of our current implementation, only installed functions can be used inside the branches. Particularly, neither functions in the same module nor functions from other modules of the same package can be used. This is not a limitation of our approach, but the current implementation.

9.2 Further research

In this work, we were exploring sensitive branching with the combination of the Spar framework. This focuses on tracking sensitivities by tracking distances of values. This is too weak to describe k-Means to be sensitive. We either need probability semantics or interleaving with executing queries for this. Future research could expand Spar with probability semantics and make k-Means feasible by this.

As mentioned above, taking ideas from [16] (expressing sensitivity as relational property and constructing a global expression in trace logic that is checked by an SMT solver) and combining it with a framework for sensitive computations is another line of research.

Finally, our branch operator's implementation does not depend on Spar but is a general operator for continuity checking of branchings. The Spar framework was used to develop the operator and to tell the story of this thesis. But the operator can be used beyond this context. It could be useful in combination in other contexts or framework where sensitivity or robustness is an important property. Or even in context that do only care about continuity.

10

Risk analysis and ethical considerations

This work's results are meant to be used in connection with large amount of sensitive data. We will asses the risks that arise in the context of applying the results.

The main risks in connection with this thesis lay in exposing insights about individuals or giving a misleading or wrong sense of privacy.

Privacy as understood in Differential Privacy and in this thesis is a likelihood of revealing insights. In particular, when employing differentially private methods, by design, no sharp boundary between "protected" and "leaked" data can be drawn, but is a continuum. This particularity needs to be understood and communicated properly before applying Differential Privacy methods.

Besides this possibility of gaining little insights by design, insights about individuals can be leaked when our method is not implemented properly. Great effort has thus been taken in order to reduce the likelihood of such an incident to the minimum. This is supported by the use of Haskell and it's strong type system. This thesis's topic arises in the context of exploring how sensitivities of queries can be inferred on the type level in order to minimize the possibilities of wrongly applying Differential Privacy.

Similarly, there could be theoretical flaw in the methods presented. This risk could be reduced by employing a proof assistant.

Finally, this work is based on Differential Privacy, a involved method for handling private data. If not employed properly, more insights can be gained than designed. In particular, Differential Privacy comes with the concept of a privacy budget that has to be taken track of.

When considering our results from an ethical perspective, these risks have be considered. Yet, Differential Privacy was invented due to the need of a trade-off between providing interesting insights and protecting individuals' data. We believe that Differential Privacy is the best suited method for this as for now while it has to be kept in mind that no analyses are possible without giving out small insights at the same time.

Bibliography

[1] E. L. Vesga, "Language-Based Differential Privacy with Accuracy Estimations and Sensitivity Analyses," Chalmers University of Technology, 2023, ISBN: 9789179058111. [Online]. Available: https://research.chalmers.se/en/publication/534817 (visited on 08/01/2023).

[2] F. McSherry, "Privacy Integrated Queries," p. 12, 2009.

[3] J. Reed and B. C. Pierce, "Distance Makes the Types Grow Stronger," p. 14, 2010.

[4] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce, "Linear Dependent Types for Differential Privacy," p. 14, 2013.

[5] C. Abuah, D. Darais, and J. P. Near. "Solo: A Lightweight Static Analysis for Differential Privacy." arXiv: 2105.01632 [cs]. (Oct. 13, 2021), [Online]. Available: http://arxiv.org/abs/2105.01632 (visited on 07/21/2022), preprint.

[6] S. Chaudhuri, S. Gulwani, and R. Lublinerman, "Continuity and Robustness of Programs," presented at the Cummunications of the ACM, Research Highlights, Aug. 1, 2012, pp. 107–115. [Online]. Available: https://www.microsoft.com/en-us/research/publication/continuity-robustness-programs/ (visited on 07/21/2022).

[7] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating Noise to Sensitivity in Private Data Analysis," in *Theory of Cryptography*, S. Halevi and T. Rabin, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2006, pp. 265–284, ISBN: 978-3-540-32732-5. DOI: 10.1007/11681878_14.

[8] C. Dwork, "Differential Privacy," in *Automata, Languages and Programming*, M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2006, pp. 1–12, ISBN: 978-3-540-35908-1. DOI: 10.1007/11787006_1.

[9] D. Kifer and A. Machanavajjhala, "No free lunch in data privacy," in *Proceedings of the 2011 International Conference on Management of Data - SIGMOD '11*, Athens, Greece: ACM Press, 2011, p. 193, ISBN: 978-1-4503-0661-4. DOI: 10.1145/1989323.1989345. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1989323.1989345 (visited on 09/29/2022).

[10] D. Otwani and R. A. Eisenberg, "The Thoralf plugin: For your fancy type needs," *ACM SIGPLAN Notices*, vol. 53, no. 7, pp. 106–118, Sep. 17, 2018, ISSN: 0362-1340. DOI: 10.1145/3299711.3242754. [Online]. Available: https://doi.org/10.1145/3299711.3242754 (visited on 04/26/2023).

[11] G. Barthe, M. Gaboardi, J. Hsu, and B. Pierce, "Programming language techniques for differential privacy," *ACM SIGLOG News*, vol. 3, no. 1, pp. 34–53, Feb. 17, 2016, ISSN: 2372-3491. DOI: 10.1145/2893582.2893591. [Online]. Available: https://dl.acm.org/doi/10.1145/2893582.2893591 (visited on 04/03/2023).

[12] E. Lobo-Vesga, A. Russo, and M. Gaboardi, "A Programming Language for Data Privacy with Accuracy Estimations," *ACM Transactions on Programming Languages and Systems*, vol. 43, no. 2, 2021, ISSN: 0164-0925. DOI: 10.1145/3452096. [Online]. Available: https://research.chalmers.se/en/publication/525255 (visited on 07/21/2022).

[13] J. P. Near, D. Darais, C. Abuah, *et al.* "Duet: An Expressive Higher-order Language and Linear Type System for Statically Enforcing Differential Privacy." arXiv: 1909.02481 [cs]. (Sep. 5, 2019), [Online]. Available: http://arxiv.org/abs/1909.02481 (visited on 01/19/2023), preprint.

[14] D. Winograd-Cort, A. Haeberlen, A. Roth, and B. C. Pierce, "A framework for adaptive differential privacy," *Proceedings of the ACM on Programming Languages*, vol. 1, 10:1–10:29, ICFP Aug. 29, 2017. DOI: 10.1145/3110254. [Online]. Available: https://doi.org/10.1145/3110254 (visited on 01/19/2023).

[15] E. Lobo-Vesga, "Let's not Make a Fuzz about it," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, May 2021, pp. 114–116. DOI: 10.1109/ICSE-Companion52605.2021.00051.

[16] G. Barthe, R. Eilers, P. Georgiou, B. Gleiss, L. Kovács, and M. Maffei, "Verifying Relational Properties using Trace Logic," in *2019 Formal Methods in Computer Aided Design (FMCAD)*, Oct. 2019, pp. 170–178. DOI: 10.23919/FMCAD.2019.8894277.

[17] M. Jha and S. Raskhodnikova, "Testing and Reconstruction of Lipschitz Functions with Applications to Data Privacy," *SIAM Journal on Computing*, vol. 42, no. 2, pp. 700–731, Jan. 2013, ISSN: 0097-5397, 1095-7111. DOI: 10.1137/110840741. [Online]. Available: http://epubs.siam.org/doi/10.1137/110840741 (visited on 01/19/2023).

A

Soundness of cswp revisited

In the Section 5, we have argued that the definition of cswp is sound. I.e. the use of the branching operator does not break sensitivity. This was based on looking at the program code and convincing oneself about specific behavior. Since it is easy to miss details, this section introduces several intermediate steps in the reasoning.

The general approach is the same. Decision boundaries lead to equality classes. For every equality class, we perform a check on a single representative of the class and argue that the branches agree.

In Chapter 5 we then argued by looking at the code that all representatives of an equality class behave similarly in the branches and post-processing. This is not obviously true. We will introduce concepts and intermediate lemmas helping us to reason about how it is enough to check one representative, i.e. that all element of an equality class behave the same.

Making "same behavior" crisp. After we have checked that the branches are equal for at least one value in all the equality classes which constitute the decision boundary, we want to formalize and exploit that all values in an equality class behave identically.

We will introduce a notion of extensional equality[1] for `Diff`s that will help us to do this step. Two Diffs are considered equal if they represent the same difference:

```
eqI d1 d2 = evaluateD d1 == evaluateD d2
```

Lemma 2. *`eqI` is extensional in the notion that every function processing equal `Diff`s and yielding the same type, evaluates to equal results. Given* $d1$ *`:: Diff dx1 dy1 c1`,* $d2$ *`:: Diff dx2 dy2 c2`*

$$d1 =_{eqI} d2 \Rightarrow \forall r(f :: (\forall dx, dy, c : \texttt{Diff dx dy c -> r})) : f\ d1 = f\ d2$$

This property matches the intuition that differences cannot be tampered inside the branches. Note that the result type `r` does not depend on `dx, dy` or `c`. Later, we will use this lemma with `r` instantiated to `Bool`.

[1] Extensional or behavioral equality means that equality is defined by external properties, in particular their behavior. In our case, the two differences (12-7) and (5-0) are constructed differently, but behave exactly identical and considered equal.

Proof. Let $(\texttt{d}_1 :: \texttt{Diff dx1 dy1 c1}), (\texttt{d}_2 :: \texttt{Diff dx2 dy2 c2})$ with $d_1 =_{eqI} d_2$ and $(f :: (\forall dx, dy, c : \texttt{Diff dx dy c -> r}))$ be a function. We will investigate every possibility how `f` can process its input and will see that d_1 and d_2 will behave equally. `f` can only interact using the public API of `Diff`.

- `f` cannot destruct `Diff` because the constructor is not exported.
- If `f` doesn't process its argument at all, it will behave equally, independent of its input.
- `f` could call `toInt`. From the context, we know that `toInt d1 == toInt d2`.
- `f` could call `cloneToNullSum`. The result is a difference that evaluates to zero. So `toInt (cloneToNullSum d1) == toInt (cloneToNullSum d2)` and we can use this proof recursively.
- `f` could call `evaluateD`. From the context, we know that `run id (evaluateD d1) = toInt d1 == toInt d2 = run id (evaluateD d1)`. So also the results after calling evaluateD will be considered equal.
- `f` could try to call `diffAddBack`. This is the most interesting case since the result would leak the first argument, resulting in non-equal behavior. But `f` need another argument of type `Subtrahend dy c` in order to call `diffAddBack`. Values of this type can only be obtained through `createRDiff` and `RDiff`s. This means type parameter `c` cannot be controlled.[2] So the argument cannot be created inside of `f` since type parameters `c` have to match while `f` being polymorphic in `c`. The extra argument neither can be passed polymorphicly from the outside since the resulting type `d` is non-polymorphic. So `f` has no chance to match the type `c` and thus cannot call `diffAddBack`.[3]

□

That brings us one step further. Let's reconsider our proof from Chapter 5 and include the detour about the checked values.

$$\begin{aligned}
\texttt{cswp}\ p_1 &= (\texttt{post . b1 . pre})\ p_1 && \text{(A.1)}\\
&\sim_{m1} (\texttt{post . b1 . pre})\ p_3 && \text{(A.2)}\\
&= (\texttt{post . b1 . pre})\ (0,0) && \text{(A.3)}\\
&= (\texttt{post . b2 . pre})\ (0,0) && \text{(A.4)}\\
&= (\texttt{post . b2 . pre})\ p_3 && \text{(A.5)}\\
&\sim_{m2} (\texttt{post . b2 . pre})\ p_2 && \text{(A.6)}\\
&= \texttt{cswp}\ p_2 && \text{(A.7)}
\end{aligned}$$

Now (A.4) is almost what we check at compile-time. And we would like to use extensionality (Lemma 6) twice, at (A.3) and (A.5). Unfortunately, this does not

[2] It could be constructed by unsafe `undefined`, but `diffAddBack` will fail on runtime in this case.

[3] Note that, if `f` would also have been polymorphic in the output-type, we could have called `diffAddBack` and our requirement would not hold.

work out with the typing. Applying the lemma here would instantiate function `f` in the lemma to `f = post . b1 . pre` which has type `Dist n (Int, Int) -> Dist n (Int, Int)`. So `f` would be polymorphic in its return type in the distances, which is not allowed in the lemma.

Detour with Extensional equality on `RTuples` Instead, we have to take the detour over another extensional equality on the result values.

```
eq0 (RTuple d11 d12) (RTuple d21 d22) =
    evaluateD d11 == evaluateD d21 && evaluateD d21 == evaluateD d22
```

Lemma 3. *`eq0` is extensional in the notion that every function processing equal `RTuple`s of the same type, evaluates to equal results. Given* $rt1, rt2$ *`:: RTuple n dy c`*

$$rt1 =_{eqO} rt2 \Rightarrow \forall r(f :: \texttt{RTuple n dy c -> r}) : f\ rt1 = f\ rt2.$$

Note that there is no polymorphism in the distances and certificates this time.

Proof. Let `rt1 = RTuple d11 d12, rt2 = RTuple d21 d22` be two `RTuples n dy c` with $rt1 =_{eqO} rt2$ and `f :: RTuple n dy c -> r` a function. `RTuple` just wraps two `Diff`s which are accessible via the constructor. Equality on `RTuple`s is inherited from equality on the inner `Diff`s. So $d11 =_{eqI} d21$ and $d21 =_{eqI} d22$. We can reason in the same way as before that the behavior of `f` must be the same for `rt1` and `rt2`. The only exception is the reasoning for `diffAddBack` where we have argued that `f` must be flexible in `c`.

This time, parameter `c` is fixed and thus `diffAddBack` could be called by `f`. We will argue, that the result of such a call must be the same. First, we will argue that all inner `Diff`s adhere to the same subtrahend. Then, we will see that also the minuends must be pairwise equal.

`rt1, rt2` and their inner diffs are bound to the same fixed type parameter `c`, called certificate. We do a case distinction on how the inner `Diff`s were constructed. Values of this type can only be obtained by `createRDiff`, `creatDiff` or `cloneToNullSum`.

If a difference were created using `createDiff`, `c` must equal `()`. For calling `diffAddBack`, an extra parameter of type `Subtrahend dy c` is needed. Values of this type can only be obtained by `creatRDiff`. Since `c` is not linked to any payload, it cannot be unified with other types than the same `c` from the same tuple. So there cannot exist a value of type `Subtrahend dy ()` to call `diffAddBack` with and the results will be functions that cannot be processed further. So same behavior for all `Diff`s and thus both `RTuple`s.

If differences were obtained from destructing `RDiff`s, every difference gets its own certificate `c` that cannot be unified with any other difference. This means, all such differences have the same minuend and subtrahend.

If differences were obtained using `cloneToNullSum`, the subtrahend and the certificate `c` are kept intact, while the minuend is replaced. This means, all such differences have the same subtrahend.

So all inner differences have the same subtrahend. Additionally, we know that $d11 =_{eqI} d21$ and $d21 =_{eqI} d22$, so they are representing the same differences. Knowing that their subtrahends match, we can conclude that also their minuends match.

Now, we have all the pieces to reason that also calling `diffAddBack` will result in the same behavior: It returns the minuend, which we have shown to be equal. □

Rounding up The last definitions necessary are

```
g1 :: forall dx dy c => Diff dx dy c -> Bool
g1 diff = (b1 diff) eq0 (b1 $ pre p3)
g2 :: forall dx dy c => Diff dx dy c -> Bool
g2 diff = (b2 diff) eq0 (b2 $ pre p3)
```

We conclude

```
True
-- computation
= g1 (pre p3)
-- by extensionality of eqI
= g1 (pre (constant 0 :*: constant 0))
-- by definition
= (b1 $ pre (constant 0 :*: constant 0)) eq0 (b1 $ pre p3)
```

I.e.

$$\texttt{b1 (pre (0,0))} =_{eqO} \texttt{b1 (pre p3)}$$

and similarly

$$\texttt{b2 (pre (0,0))} =_{eqO} \texttt{b2 (pre p3)}.$$

Time to harvest:

$$\begin{aligned}
&\texttt{b1 (pre p3)} && \text{(A.8)}\\
=_{eqO}\ &\texttt{b1 (pre (0,0))} && \text{(A.9)}\\
=_{eqO}\ &\texttt{b2 (pre (0,0))} && \text{(A.10)}\\
=_{eqO}\ &\texttt{b2 (pre p3)} && \text{(A.11)}
\end{aligned}$$

Equality (A.10) now is what is checked at compile time. The other equalities were derived just before from the extensionality of `eqI`. Our final derivation now becomes

$$\begin{aligned}
\texttt{cswp}\ p_1 =&\texttt{(post . b1 . pre)}\ p_1 && \text{(A.12)}\\
\sim_{m1}&\texttt{(post . b1 . pre)}\ p_3 && \text{(A.13)}\\
=&\texttt{(post . b2 . pre)}\ p_3 && \text{(A.14)}\\
\sim_{m2}&\texttt{(post . b2 . pre)}\ p_2 && \text{(A.15)}\\
=&\texttt{cswp}\ p_2 && \text{(A.16)}
\end{aligned}$$

where (A.14) is by extensionality of `eq0`. With this, we are finally done: We have shown that `cswp` $p_1 \sim_n$ `cswp` p_2 and that cswp is sensitive with sensitivity 1!

In the appendix B, we will argue that this line of reasoning can be generalized, and four requirements are synthesized that, if met, guarantee soundness of branching.

B

Soundness of branch

In the Section 5, we have argued that branching in our cswp implementation does not break sensitivity. In Appendix A, we have revisited this proof and added fine-grained steps of reasoning where we just looked at the code before. In the following paragraphs, we will take a step back and generalize our arguing to other branchings over sensitive values. This will be along the same lines as Chapter A, and we focus on the generalization. For greater detail of explanation and development of the idea, the reader is referred to section 5. Requirements for the guarantees to hold generally are investigated.

The branch operator allows arbitrary Haskell expressions that are closed given the condition variables. Yet, the operator needs some structure on the types of the variables in the condition and the type of the branch results.

B.1 Requirements

The variables need to map to `Int`s: Firstly, when evaluating the condition at run time, numeric values have to computed from the variables. For this, we introduce a typeclass `ToInt`.

```
class ToInt tv where
    toInt :: tv -> Int
```

The branch operator expects an instance of this typeclass to be available for the type of variables. Secondly, for asserting agreement at a specific boundary value, variables have to be instantiated from `Int`s. The first parameter of branch thus is expected to contain an appropriate function.

The results of the branches need to be comparable in order to assert agreement. For this, we introduce a typeclass `BranchResult`.

```
class BranchResult t1 t2 where
    eq0 :: t1 -> t2 -> Bool
```

The branch operator expects instances of this class to be available for the results.[1]

[1]Note that, we are allowing some polymorphism in the result. This is because the branch results are polymorphic (in our cases in the distance and certificates). For the later formal reasoning, we need `eq0` to be defined for different type variables. When evaluated on the computer, both type

In which cases will the use of branch guarantee sensitive branching? Four conditions have to hold.

`eq0` is an equality relation. This means, it has to be reflexive, transitive and symmetric.

`eq0` implies same behavior. This is described by the concept of extensional equality `eq0 :: t -> t -> Bool`. Given two branch results $\mathtt{r}_1, \mathtt{r}_2 :: \mathtt{t}$, when they are considered equal, every function should behave equally on them:

$$\mathtt{r}_1 =_{\mathtt{eq0}} \mathtt{r}_2 \Rightarrow \forall \mathtt{p}\ (\mathtt{f} :: \mathtt{t} \to \mathtt{p}) : \mathtt{f}\ \mathtt{r}_1 = \mathtt{f}\ \mathtt{r}_2$$

Note, that this property only is valid for comparisons of same type.

`toInt` induces equalityclasses of same behavior on condition variables. `toInt :: t -> Int` naturally induces an equality measure:

```
eqI :: t1 -> t2 -> Bool
eqI v1 v2 = (toInt v1) == (toInt v2)
```

Condition variables can be polymorphic. Let `tv :: * -> *` be the constructor for condition variables with instances `ToInt (tv t)`, $\forall$`t :: *` available.

Now, *same behavior* is again formalized using the concept of extensional equality. This time, we add some relaxations and allow fine-grained polymorphism. Given two values $v_1 \in tv\ t1, v_2 \in tv\ t2$, when they evaluate to the same number, every family of functions should behave equally on them.

$$v_1 =_{\mathtt{eqI}} v_2 \Rightarrow \forall \mathtt{p}\ (\mathtt{f} :: \forall \mathtt{c}.\mathtt{tv}\ \mathtt{cp}) : f\ v_1 = f\ v_2$$

`toInt` and `fromInt` reverse each other modulo equality.

$$\forall \mathtt{v} : \mathtt{fromInt}\ (\mathtt{toInt}\ \mathtt{v}) =_{\mathtt{eqI}} \mathtt{v}$$

B.2 How these guarantee soundness

Finally, we can reason on an abstract level that the usage of the branch operator given those requirements is sound and does not break the Spar-calculus.

Given a program employing a branching.

```
main :: Dist dIn tIn -> Dist (l*dIn) tOut
main = post . b . pre

pre :: Dist dIn tIn -> ti dIn tIn
post :: tr dIn tIn -> Dist (l*dIn) tOut

b :: ti dIn tIn -> tr dIn tIn
b b_inp = $(branch fromInt c b_1 b_2)
```

variables will be the same.

Here `fromInt`, `c`, $\texttt{b}_1$, $\texttt{b}_2$ are meta-variables. In the real code, they will be replaced by quasiquotes. For this reasoning, we will assume that the semantics of those variables are available via functions

```
fromInt :: Int -> tv dEval tEval
c :: ti d t -> Bool
b_1, b_2 :: ti d t -> tr d t
```

Note the difference between types `tv` of condition variables and `ti` branch inputs. We can think of `ti` being a vector of `tv`s.

The idea is that every branch usage involving Spar-values includes some preprocessing transforming the Spar-values into some other type `ti`. The variables occurring in the condition may be of the same type `ti` or some related type `tv` when `ti` is deconstructed in the function definition. The branches then transform the values of type `ti` into values of type `tr`. Finally, the result is transformed back to Spar-values.

Given $p1 \sim_{dIn} p2$, we can distinguish two cases:

c (pre p1) = c (pre p2). In that case, applying $p1$ or $p2$ to the branching will result in the same branch being executed. Since the branches are not breaking the Spar-calculus, we get sensitivity of `main` for free.

c (pre p1) ≠ c (pre p2). This means, for one value, one branch is executed, and for the other, the other branch. For this proof, we think of a continuation of Spar-values and our function for the real numbers. Let p3 be a value between p1 and p2 on the decision boundary such, that $p1 \sim_{m1} p3$, $p3 \sim_{m2} p2$ and $dIn = m1 + m2$. Let `p3' = fromInt (toInt p3)` From the last requirement, we know: $\texttt{p3}' =_{\texttt{eqI}} \texttt{p3}$. Define

```
g1 :: t -> Bool
g1 inp = (b1 $ pre inp) eq0 (b1 $ pre p3)
g2 :: t -> Bool
g2 inp = (b2 $ pre inp) eq0 (b2 $ pre p3)
```

Using extensional equality of eqI, we can conclude

```
  True
= g1 p3                    -- computation
= g1 p3'                   -- extensionality of eqI
= (b1 $ pre p3') eq0 (b1 $ pre p3)   -- definition
```

This means `b1 (pre p3)` $=_{\texttt{eq0}}$ `b1 (pre p3')`, similarly `b2 (pre p3')` $=_{\texttt{eq0}}$ `b2 (pre p3)` and further

$$\begin{aligned} &\texttt{b1 (pre p3)} && \text{(B.1)}\\ =_{eqO}\ &\texttt{b1 (pre p3')} && \text{(B.2)}\\ =_{eqO}\ &\texttt{b2 (pre p3')} && \text{(B.3)}\\ =_{eqO}\ &\texttt{b2 (pre p3)}. && \text{(B.4)} \end{aligned}$$

The first and last equality were derived before. Equality (B.3) is checked at compile-time.

We can now conclude

$$\begin{aligned}
&\texttt{main p1} && \text{(B.5)}\\
=&(\texttt{post . b . pre})\ \texttt{p1} && \text{(B.6)}\\
=&(\texttt{post . b1 . pre})\ \texttt{p1} && \text{(B.7)}\\
\sim_{m1*l}&(\texttt{post . b1 . pre})\ \texttt{p3} && \text{(B.8)}\\
=&(\texttt{post . b2 . pre})\ \texttt{p3} && \text{(B.9)}\\
\sim_{m2*l}&(\texttt{post . b2 . pre})\ \texttt{p2} && \text{(B.10)}\\
=&(\texttt{post . b . pre})\ \texttt{p2} && \text{(B.11)}\\
=&\texttt{main p2} && \text{(B.12)}
\end{aligned}$$

Equality (B.9) is derived from extensional equality of `eq0` and the previous paragraph.

Finally, we have shown that $|\texttt{main p1} - \texttt{main p2}| \leq l * (m1 + m2) = l * dIn$.

We can see that the requirements and branching as an operation are not linked to the Spar framework at all. We expect this operator to be useful for checking continuity in other contexts as well with the same requirements and adapted reasoning.

C

More details on Mergesort

C.1 Closer look at decision function

The decision function is used while merging to determine which of the two tips to move next. The following types facilitate the process:

```
type DecFun = forall c. CDist c -> MergeDec c
data CDist c = forall d. CDist (Dist d Int)
data MergeDec c = L (CDist c) | R (CDist c)
```

From these types, the behavior of the decision function is very limited. In the following paragraphs, we will establish useful properties. Those will be handy to prove correctness of our implementation, and we will see that they are also necessary for sensitivity.

Property 1. *The decision function is deterministic and constant for the whole sorting process.*

By being a pure function, it must behave deterministically. Constance follows from the usage of the function in `mergeSort'` and `merge`. This is necessary for arguing that a specific next step matches the ongoing process.

Given a decision function `cmp`, we define $a \prec b$ iff the following evaluates to `True`:

```
case (cmp (CDist $ (number a) .- (number b)), cmp (CDist $ (number 0))) of
    (L _, L _) = True
    (R _, R _) = True
    _          = False
```

a is called smaller than b, iff $a \prec b$ and we define $a \doteq b \Leftrightarrow a \prec b \wedge b \prec a$. Before we show that $\prec$ in fact is a preorder, we establish some other properties.

Property 2. $\prec$ *is translation-invariant.*

The only parameter the decision function can base its result on, is of type `CDist`. Since `CDist` only carries a Spar-value instanced to the difference between the elements, $\prec$ cannot behave differently for translated values. Thus $\forall c \in \mathbb{Z}$ if $a \prec b$ then also $a - c \prec b - c$.

Property 3. *The behavior can only change at similar values.*

More precisely: if $a \prec b$, but $a+1 \not\prec b$, then $b = a \vee b = a+1$.

Proof. Given a function `cmp :: DecFun` that is not breaking the Spar calculus. Let `a, b :: Int` with $a \prec b$ and $a+1 \not\prec b$. We use `cmp` to define a sensitive function

```
f :: Dist d (Int, Int) -> Dist d Int
f (v1 :*: v2) = case cmp (v1 .- v2) of
                    L diff -> diff
                    R diff -> constant 0 .- diff
```

and define `va::Dist 1 Int` with semantics $a \sim_1 a+1$ and `vb::Dist 0 Int` with semantics $b \sim_0 b$. It holds `va :*: vb :: Dist 1 (Int, Int)` with semantics $(a,b) \sim_1 (a+1,b)$. By definition of $\prec$ and the way we picked a and b, we can reason that `f (va :*: vb)` has semantics $(a-b) \sim (b-(a+1))$. The distance between those values is $d' = |2(a-b)+1|$. Because `cmp` is not breaking sensitivity, we already know that $d' \leq 1$ and thus $b = a \vee b = a+1$ must hold. □

We will use the following generalization several times:

Lemma 4 (Midpoint.)**.** *Given $a \prec b$ and $a \not\prec c$, there is $\tau \in [0,1]$ with $a = b+\tau(c-b)$ and $\tau(c-b) \in \mathbb{Z}$.*

Similarly, given $a \prec b$ and $c \not\prec b$, there is $\theta \in [0,1]$ with $b = a+\tau(c-a)$ and $\tau(c-a) \in \mathbb{Z}$.

Proof. Note that $b \neq c$. Define $\tau' = \frac{1}{|c-b|}$. Let $k \in [1,\ldots,|c-b|]$ be the minimal with $a \not\prec b + k * \tau'(c-b)$. Now $a \prec b + (k-1) * \tau'(c-b)$ and $a \not\prec b + k * \tau'(c-b)$. With the Property 3, we conclude $a = b + (k-1)\tau'(c-b) \vee a = b + k\tau'(c-b)$. We set $\tau = (k-1)\tau'$ or $\tau = k\tau'$ such that $a = b + \tau(c-b)$ holds.

The second part uses exactly the same steps. □

Property 4. *$\prec$ is a preorder.*

Proof. ($\prec$ is preorder.) 1. ($\prec$ is reflexive). This holds by definition.

2. ($\prec$ is transitive). Consider a,b,c with $a \prec b$ and $b \prec c$, but $a \not\prec c$. We will show that this is impossible. Note that $a \neq b$, $b \neq c$ (choice of a,b,c) and $a \neq c$ (reflexivity). With lemma 4, we define τ, θ with $a = b+\tau(c-b)$ and $c = b+\theta(a-b)$. Plugging the first in the second yields $c = b+\theta((b+\tau(c-b))-b) = b+\theta\tau(c-b)$. We can conclude that either $b = c$ or $\theta = \tau = 1$ implying $c = a$. Both contradict a,b,c being inequal. Hence this cannot occur. □

We show additional lemmas we need later for showing correctness.

Lemma 5.

$(\exists a,b : a \prec b \wedge a < b) \Rightarrow \forall c,d : c \leq d \Rightarrow c \prec d$

and

$(\exists a,b : a \prec b \wedge a > b) \Rightarrow \forall c,d : c \geq d \Rightarrow c \prec d.$

Proof. Let w.l.o.g. $a < b$ (other direction analogue). We will show that $a \prec a+1$. Assume $a \not\prec a+1$, we have $a \prec b$ and $a \not\prec a+1$. With Lemma 4 there is θ with $a = a + 1 + \theta(b - a - 1)$. Thus $1 = \theta(1 + a - b)$. Since we know that $a < b$ and $1 \geq \theta \geq 0$, we can conclude $a = b$. But this contradicts the assumption.

Now we can show that $\forall c, d : c \leq d \to c \prec d$. Because $a \prec a + 1$, we can reason by translation-invariance and transitivity that $c \prec d$. □

Lemma 6. *Totality implies* $\leq \to \prec$ *or* $\geq \to \prec$.

Proof. Given a, b with $a \prec b$, but $a \neq b$ (exists due to totality). Let w.l.o.g. $a < b$. The result follows from Lemma 5. □

Lemma 7. *Totality and anti-symmetry implies* $\prec \equiv \leq$ *or* $\prec \equiv \geq$.

Proof. Consider again arbitrary a, b with $a \prec b$, but $a \neq b$ (exists due to totality). Let w.l.o.g. $a \leq b$.

Now, we'll show that $\forall c, d : c \prec d \to c \leq d$. Given $c \prec d$. If $d \leq c$, we can conclude with Lemma 6 that $d \prec c$, with anti-symmetry $c = d$ and thus $c \leq d$. □

Phew, now we can think of the decision as a comparison.

C.2 Correctness of `mergeSort'`

With those properties we can show that the invariants on `PartialMerge` hold and merging and sorting work as expected.

As a reminder, the **invariants** are

- that the two lists of remaining elements are sorted ascending wrt. $\prec$
- that the partial result is sorted inversely
- all elements in the partial result are smaller or equal than the remaining lists.

Reconsider the definition of `merge` (the only function handling `PartialMerge`s) and the following definitions of helper functions:

```
merge :: Dist d1 (Vec l1 Int) -> Dist d2 (Vec l2 Int)
  -> DecFun -> Dist (d1 + d2) (Vec (l1 + l2) Int)
merge v1 v2 f = merge' (startPartialMerge v1 v2)
    where
        merge' pm  | isMerged pm = getResult pm
        merge' pm = merge' (progressWithComparison pm f)

progressWithComparison :: PartialMerge d l -> DecFun -> PartialMerge d l
progressWithComparison pm@(PartialMerge _ (l:>ls) Nil) f =
    progressFromLeft pm
progressWithComparison pm@(PartialMerge _ Nil (r:>rs)) f =
    progressFromRight pm
progressWithComparison pm@(PartialMerge _ (l:>ls) (r:>rs)) decFun =
    case decFun $ CDist (l .- r) of
        L _ -> progressFromLeft pm
        R _ -> progressFromRight pm
```

Merging starts by creating the initial `PartialMerge` with empty partial result and the inputs as parts still to be processed. Those parts are sorted by assumed requirement to merge and we have already seen that the inputs will indeed be sorted (called in `mergeSort''`). `progressWithComparison` then processes `PartialMerge`s adhering to the invariants: If exactly one of the lists with remaining elements is empty, the other remaining elements are moved to the result list one by one. Because the remaining lists' elements are greater or equal to the partial result, the sorting in the partial result is preserved. The other invariants can be directly taken over. If both lists to be merged have remaining elements, the decision function is considered. Since we know that the decision function encodes a comparison and that the partial result and remaining parts are sorted with respect to this comparison, we can conclude that also in this case, the invariants are preserved: The new result tip is a smallest one of the two parts' tips, so smaller or equal than all the remaining elements, but larger or equal to every element of the partial result. So moving this element over to the partial result preserves the invariants.

Here we assume the induced order to be total. If this doesn't hold, it does not make sense to use it for sorting, and we do not care about the correctness of the result. In a later paragraph (Section C.4), we will show that this cannot break sensitivity.

C.3 Soundness of `mergeSort`

The final definition of mergeSort is

```
df :: DecFun
df diff = $(branch [| \i -> (branchCDist . constant) i |]
           [| diff > 0|]
           [| goL diff |]
           [| goR diff |]
         )
mergeSort :: Dist d (Vec l Int) -> Dist d (Vec l Int)
mergeSort v = mergeSort' v df
instance BranchResult (MergeDec c1) (MergeDec c2) where
  eq (L diff1) (R diff2) = toInt diff1 == 0 &&  toInt diff2 == 0
  eq (R diff1) (L diff2) = toInt diff1 == 0 &&  toInt diff2 == 0
  eq _ _ = True
```

We will now show that our usage of the branch operator is sound by showing that the general requirements developed in Appendix B are met. For doing so, we establish another extensional equality on `PartialMerge`.

Remember, the order $\prec$ induces an equality $\doteq$ by $a \doteq b :\Leftrightarrow a \prec b \wedge b \prec a$.

We define equality on `PartialMerge` and show extensionality of it. Two `PartialMerge`s are considered to be equal iff their partial results equal wrt. $\doteq$ and the elements contained in their remaining parts equal (again wrt. $\doteq$). This is, we do not care in which remaining part elements are stored, but the multi-sets of both parts must equal. Since `PartialMerge` is not publicly visible, it is enough to reason about the actual usages; they are only processed in `progressFromLeft` and `progressFromRight`.[1]

Lemma 8. *`merge'` turns equal `PartialMerge`s as input into equal lists as result.*

Proof. Let's call the inputs pmi_1 and pmi_2. The proof works by induction on the number of elements in the remaining lists.

- Let's first consider the case, that pmi_1 has no remaining elements. This implies the same for pmi_2, because it is equal to pmi_1. So the result of `merge'` for both inputs will be their respective heads, which are known to be equal and sorted wrt. $\prec$.

  Assuming (and verifying) that our comparison function is anti-symmetric, we can even conclude that the result is sorted wrt. to $\geq$ or $\leq$, depending on the fixed comparison function. So $\doteq \equiv =$ and the resulting vectors must be equal.

- Let's now consider that there are some elements left.

  A new `PartialMerge` will be constructed by `progressWithComparison` and passed to `merge'` recursively. We will show that also the newly constructed `PartialMerge`s equal one another and use induction to conclude that the resulting lists will be equal.

  Consider `progressWithComparison` and two equal inputs pmi_1, pmi_2. In all the cases a smallest element of the remaining elements is appended to the

[1] If `PartialMerge` would be visible from outside, equality on it would not be extensional.

head. Since the smallest elements in the remaining elements of pmi_1 and pmi_2 must equal, also the elements taken over to the result will equal. The same holds for the remaining elements: Probably not the same smallest element was moved, but since we only consider the remaining elements as a multi-set, the resulting `PartialMerge`s are equal. So `progressWithComparison` translates equal inputs into equal results.

So the `PartialMerge`s used in the recursive call are equal and by induction thus the results.

□

We have seen that `merge'` transforms equal `PartialMerge`s into equal sensitive lists. `merge` called on equal input lists calls `merge'` with equal `PartialMerge`s. Equality on sensitive lists is extensional, and we conclude, that equality on `PartialMerge` also is extensional.

Now, it can be shown that we meet all the necessary requirements for the use of branching as defined in Section B:

eqO is equality relation. Because eqO is based on another equality relation, the necessary properties are inherited

eqO is extensional. Consider equal `MergeDec c` values. Note that this value is certified. The only way to produce equal `MergeDec c` values according to the same certificate is by using `goL` or `goR` on the exact same `CDist`. This implies that the payload of `MergeDec c` comes from the same pair. With this in mind, equal `MergeDec c` must agree on the payload, but not necessarily on the direction to go. Looking at the definition of `progressWithComparison`, we can conclude that equal `MergeDec` result in equal `PartialMerge`. `eqO` inherits `PartialMerge`'s equality's extensionality.

Equality by toInt is extensional The type of the branching variable is `CDist` and as we have seen, it just wraps a Spar-value. This value is accessible via `toInt`. So if for two `CDist`-values `cd1, cd2 toInt cd1 == toInt cd2` holds, the values of `cd1, cd2` are already equal in every possible way.

toInt and fromInt reverse Looking at the definition, it is easy to verify that both function reverse each other.

Hence our usage of the branch operator is sound and Mergesort is sensitive.

Note that we require an encapsulated decision function for several reasons. First, in order for the invariants to hold, we have to assume a constant decision function. Second and most interestingly, note that the trace of comparisons can also change a lot for slightly changed inputs. By requiring a pure capsuled decision function, we prevent any leakage of information from the comparisons. At least in safe Haskell.

C.4 Requirements necessary and sufficient for sensitivity

Our reasoning for sensitivity made use of the requirements we developed for the branch operator and thus solely deal with equalities. For this, we established many invariants and properties. In this section, we will take a step back and show that many of these requirements are not only useful for sufficiently reasoning sensitivity, but are also necessary. A few ways that the decision function or merging procedure could misbehave are examined with respect to the implications on sensitivity and correctness.

The other tip is stored at the appropriate part tip. If `progressFromLeft` or `progressFromRight` would put the remaining tip on top of the wrong list, we would not only break correctness of Mergesort, but also sensitivity. Consider the list $[1,4,5,1,2,3]$ and the progressing functions putting the other candidate always on the wrong tip. The decision function favors the first part's head in case of equality. This will split the list into $[1,4,5]$ and $[1,2,3]$ to be sorted. The miss-behaving merging is no problem for sorting these parts yielding the same lists again. The trace of the `PartialMerge`s show the merging process: $([],([1,4,5],[1,2,3])) \rightsquigarrow ([1],([1,4,5],[2,3])) \rightsquigarrow ([1,1],([2,4,5],[3])) \rightsquigarrow ([2,1,1],([3,4,5],[])) \rightsquigarrow ([5,4,3,2,1,1],([],[]))$. The miss-behaving merge still sorts this list as expected.

Let's consider another list with distance 1 to the previous list $[2,4,5,1,2,3]$. Again, we will trace the occurring merge: $([],([2,4,5],[1,2,3])) \rightsquigarrow ([1],([4,5],[2,2,3])) \rightsquigarrow ([2,1],([5],[4,2,3])) \rightsquigarrow ([4,2,1],([5,2,3],[])) \rightsquigarrow ([3,2,5,4,2,1],([],[]))$. This list does not look correctly sorted. And it has distance 9 to the previous result.

That is, two inputs with distance 1 result in two results with distance 9. With a miss-behaving merge, mergesorting is not 1-sensitive any longer. That is why it is important that the merge is well-behaved.

In the course of showing correctness, we assumed the decision function to be total and anti-symmetric. We'll now see that a non-total or non-anti-symmetric decision function doesn't change behavior, i.e. it always returns the same, fixed constructor. So there is no relevant branching, and it cannot break sensitivity.

Decision function is not total. We will show: $\forall c,d : c \prec d \Rightarrow c = d$. So let $c \prec d$ and w.l.o.g. $c \leq d$. This implies $n \prec n + d - c$ (translation-invariance). There are $a < b$ with $a \not\prec b$ and $b \not\prec a$ (non-totality).

Consider $a' = a + 1$. If $a \prec a'$, then by translation-invariance and transitivity also $a \prec b$ is true, which contradicts the assumption $a \not\prec b$. Thus $a \not\prec a'$.

Now we have $a \prec a + d - c$ and $a \not\prec a + 1$. So there is τ with $a = a + d - c + \tau(a + d - c - a - 1)$. We reason that $\tau = (1 - \tau)(c - d)$, so $c \geq d$ and thus $c = d$.

That means, the decision function actually identifies equality and divergent branching

if the values are equal does not break sensitivity.

Total decision function is not anti-symmetric. We'll show that $\forall a, b : a \prec b \wedge b \prec a$. Given w.l.o.g. $c < d$ but $c \prec d \wedge d \prec c$ (not anti-symmetric). We know that $a \leq b \vee b \leq a$ and can conclude with Lemma 5($a \prec b \wedge a < b \Rightarrow \leq \rightarrow \prec$) that $a \prec b \wedge b \prec a$.

That means, the decision function returns the same constant constructor.