Anti-Unification with Type Classes

Nicolas Tabareau, Éric Tanter, Ismael Figueroa

Anti-Unification with Type Classes

Nicolas Tabareau[1], Éric Tanter[2], Ismael Figueroa[1,2]

1: ASCOLA Group
INRIA, France
nicolas.tabareau@inria.fr

2: PLEIAD Laboratory
DCC, University of Chile – Chile
etanter@dcc.uchile.cl
ifiguero@dcc.uchile.cl

Abstract

The anti-unification problem is that of finding the most specific pattern of two terms. While dual to the unification problem, anti-unification has rarely been considered at the level of types. In this paper, we present an algorithm to compute the least general type of two types in Haskell, using the logic programming power of type classes. That is, we define a type class for which the type class instances resolution performs anti-unification. We then use this type class to define a type-safe embedding of aspects in Haskell.

1. Introduction

The anti-unification problem—as first been considered independently by Plotkin [17] and Reynolds [19]—is that of finding the most specific template (pattern) of two terms. It is dual to the well-known unification problem, which is the computation of the most general instance of two terms. In Plotkin's seminal paper, the need for anti-unification is justified from a logical point of view. The question to be solved was how to generalize the following clauses automatically:

```
The result of heating this bit of iron to 419°C was that it melted.
The result of heating that bit of iron to 419°C was that it melted.
--------------------------------------------------------------------
The result of heating any  bit of iron to 419°C was that it melts.
```

This is formalized in Plotkin's paper as:

```
BitofIron(bit 1) ∧ Heated(bit 1, 419) ⊃ Melted(bit 1)
BitofIron(bit 2) ∧ Heated(bit 2, 419) ⊃ Melted(bit 2)
------------------------------------------------------
(x) BitofIron(x) ∧ Heated(x, 419) ⊃ Melted(x)
```

While unification is a common tool in the definition of type inference algorithms, anti-unification has rarely been considered at the level of types. However, the very same generalization can be done if `BitofIron`, `Heated` and `Melted` are seen as type constructors `A`, `B` and `C`, and `bit1`, `bit2` and `419` as types t_1, t_2, and t_3.

```
A(t₁) → B(t₁,t₃) → C(t₁)
A(t₂) → B(t₂,t₃) → C(t₂)
-------------------------
∀ a, A(a) → B(a,t₃) → C(a)
```

To the best of our knowledge, there are only two pieces of work in this area[1]. One paper of Pfenning on the unification and anti-unification in the calculus of construction (CoC) [15] advocates for anti-unification as a mean to generalize proofs. But we are aware of no implementation of this technique in a proof assistant based on CoC.

[1]There is also the work Alpuente et al.[1] on anti-unification for typed terms, but the generalization is at the level of terms and not of types.

Another group of papers on AspectML [4], an aspect oriented extension of ML, uses an anti-unification algorithm to type-check pointcuts of an aspect. Indeed, aspects provide the facility to intercept the flow of control in an application and perform new computations. In this approach, computation at certain execution points, called *join points*, may be intercepted by a particular condition, called *pointcut*, and modified by a piece of code, called *advice*, which is triggered only when the runtime context at a join point matches the conditions specified by a pointcut. To be safe, the type of a pointcut must specify the type of join points it may match—which we call its *matched type*. But in case those join points are of different types, the matched type of the pointcut is not the unifier of those types, but rather the least generalization.

In this paper, we propose to use the type class system of Haskell to perform anti-unification during type class resolution. Indeed, because type class resolution somehow performs logic programming, it is possible to write an algorithm at the level of types using type classes and type class instances in the same way as we can write an algorithm in Prolog using relations and (Horn) clauses. More precisely, we define a type class `LeastGen` for which an instance `LeastGen a b c` is valid iff `c` is the least generalization of `a` and `b`. For instance, the following Haskell code computes a generalization similar to Plotkin's example[2]:

```
data BitofIron a = BitofIron a
data Heated a b = Heated a b
data Melted a = Melted a
data Bit1
data Bit2

bit1 :: BitofIron Bit1 → Heated Bit1 Int → Melted Bit1
bit1 = undefined

bit2 :: BitofIron Bit2 → Heated Bit2 Int → Melted Bit2
bit2 = undefined

generalize :: LeastGen t1 t2 t3 ⇒ t1 → t2 → t3
generalize = undefined
```

That is, Haskell type class resolution is able to derive automatically the type of `generalize bit1 bit2` as

```
(generalize bit1 bit2) :: Bitofiron a → Heated a Int → Melted a
```

After defining formally anti-unification in the setting of the Haskell type system, we present the type classes responsible for computing the least general type of two types (Section 2) and prove its correctness. To illustrate the potential of the anti-unification type class, we then present an embedding of aspects in Haskell (Section 3), where type safety crucially relies on the use of the `LeastGen` type class (Section 4).

2. Anti-unification with Type Classes

We start by briefly summarizing the notion of type substitutions and the *is less general* relation between types. Then we describe a novel anti-unification algorithm implemented with type classes, on which the type class `LeastGen` is based. The algorithm relies on the fact that a multi-parameter type class `R` $t_1 \dots t_n$ can be seen as a *relation* `R` on types $t_1 \dots t_n$, and **instance** declarations as ways to (inductively) define this relation, in a manner very similar to logic programming. Note that we do not consider type class constraints in the definition (see Section 2.6 for a discussion).

2.1. Least General Type

In this section we summarize the definition of type substitutions and introduce formally the notion of least general type in a Haskell-like type system (without ad-hoc polymorphism). Thus, we have types $t ::= Int, Char, \dots, t_1 \rightarrow t_2, T\ t_1\ \dots\ t_m$, which denote primitive types, functions, and m-ary type constructors , in addition to user-defined types. We consider a typing environment $\Gamma = (x_i\colon t_i)_{i \in \mathbb{N}}$ that binds

[2] Note that in our example, the computational content of functions is not relevant, so we use `undefined` to inhabit each type.

variables to types.

Definition 1 (Type Substitution, from [16])**.** A type substitution σ is a finite mapping from type variables to types. It is denoted $[x_i \mapsto t_i]_{i\in\mathbb{N}}$, where $dom(\sigma)$ and $range(\sigma)$ are the sets of types appearing in the left-hand and right-hand sides of the mapping, respectively. It is possible for type variables to appear in $range(\sigma)$.

Substitutions are always applied simultaneously on a type. If σ and γ are substitutions, and t is a type, then $\sigma \circ \gamma$ is the composed substitution, where $(\sigma \circ \gamma)t = \sigma(\gamma t)$. Application of substitution on a type is defined inductively on the structure of the type.

Substitution is extended pointwise for typing environments in the following way: $\sigma(x_i\colon t_i)_{i\in\mathbb{N}} = (x_i\colon \sigma t_i)_{i\in\mathbb{N}}$. Also, applying a substitution to an expression e means to apply the substitution to all type annotations appearing in e.

Definition 2 (Less General Type)**.** We say type t_1 *is less general* than type t_2, denoted $t_1 \preceq t_2$, if there exists a substitution σ such that $\sigma t_2 = t_1$. Observe that $\preceq$ defines a partial order on types (modulo α-renaming).

Definition 3 (Least General Type)**.** Given types t_1 and t_2, we say type t is the *least general type* iff t is the supremum of t_1 and t_2 with respect to $\preceq$.

2.2. Direct Functional Algorithm

In his thesis [7], Huet has shown that the computation of the least generalization can be defined functionally. This algorithm (named λ) has since been rephrased and we present here a version of Østvold [14] that works in the same way as our type class algorithm. The idea is to compute the least generalization of two terms t and u recursively by computing at the same time the current (injective) substitution and the least generalization as follows:

$$
\begin{array}{rcll}
\lambda(t,t,\theta) & = & (t,\theta) & \\
\lambda(f(t_1,\ldots,t_n), f(u_1,\ldots,u_n),\theta_0) & = & (f(x_1,\ldots,x_n),\theta_n) & \text{where } \lambda(t_i,u_i,\theta_{i-1}) = (x_i,\theta_i) \\
\lambda(t,u,\theta) & = & (x,\theta) & \text{if } \theta(x) = (t,u) \\
\lambda(t,u,\theta) & = & (y,\theta') & \text{where } y \notin \mathit{dom}\ \theta \text{ and } \theta' = \theta + \{y \mapsto (t,u)\} \\
\\
\texttt{leastGen}(t,u) & = & \pi_1(\lambda(t,u,\{\})) &
\end{array}
$$

The algorithm λ tries to apply the rules from top to down. That is, if the two terms are equal, it returns the term and the current substitution. If the two terms share the same top constructor, it applies the generalization recursively on the arguments and collects back the result. When the two terms do not share the same top constructor, if there is already a type variable in the current substitution that relates these two terms, this variable is just returned, with the substitution. If it is not the case, a *fresh* variable is introduced and the substitution is extended accordingly. The least generalization of two terms is then obtained by applying λ with the empty substitution (and taking the first element of the resulting pair).

The correctness of this algorithm has been proved in [14]. The rest of this section presents how to compute this algorithm at the level of type classes and proves its correctness in this setting.

2.3. Encoding Substitutions with Type Classes

As a warm up, we present an (folklore) encoding of substitutions with type classes. The basic idea is to emulate the recursive type of substitutions with a type class that represents a *recursive kind*.

```
data SubstEmpty              class Substitution s
data SubstCons x sx s        instance Substitution SubstEmpty
                             instance Substitution s ⇒ Substitution (SubstCons x sx s)
```

Thus, a substitution—*i.e.* an instance of the `Substitution` class—is either the type `SubstEmpty`, or the type `SubstCons x sx s` where `s` is a substitution.

```
1  class (Substitution σ_in, Substitution σ_out) ⇒ LeastGen' a b c σ_in σ_out | a b c σ_in → σ_out
2
3  Inductive case: The two type constructors match,
4  recursively compute the substitution for type arguments a_i, b_i.
5  instance (LeastGen' a_1 b_1 c_1 σ_0 σ_1, ...,
6            LeastGen' a_n b_n c_n σ_{n-1} σ_n,
7            T c_1 ... c_n ~ c)
8         ⇒ LeastGen' (T a_1 ... a_n) (T b_1 ... b_n) c σ_0 σ_n
9
10 Default case: The two type constructors don't match, c has to be a variable,
11 either unify c with c' if c' ↦ (a, b) or extend the substitution with c ↦ (a, b)
12 instance (Substitution σ_in, Substitution σ_out,
13           Analyze c (TVar c),
14           MapsTo σ_in c' (a,b),
15           VarCase c' (a,b) c σ_in σ_out)
16        ⇒ LeastGen' a b c σ_in σ_out
17
18 extends the substitution if required
19 class (MaybeType v, Substitution σ_in, Substitution σ_out) ⇒ VarCase v ab c σ_in σ_out | v ab σ_in → σ_out c
20 instance Substitution σ_in ⇒ VarCase None ab c σ_in (SubstCons ab c σ_in)
21 instance Substitution σ_in ⇒ VarCase (Some c) ab c σ_in σ_in
```

Figure 1: Definition of the `LeastGen'` type class. An instance holds if `c` is the least general type of `a` and `b`.

Note that this encoding is not completely satisfactory because it is *untyped*! Indeed, Haskell has a very powerful and expressive static type system, but here we want to do programming at the type level, and the kind system of Haskell is unsatisfactorily inexpressive. This issue is well known and a recent paper proposes a way to add data types and polymorphism at the level of kinds [22]. Waiting for its implementation in GHC, we have no choice but to use the untyped version of substitutions for the moment.

In the same way, we can define a class `MaybeType` that corresponds to the `Maybe` data structure but at the level of types:

```
data None              class MaybeType a
data Some a            instance MaybeType None
                       instance MaybeType (Some a)
```

Then, we can use the type class resolution mechanism of Haskell to encode a function that takes a substitution `s` and a type `x` and binds `sx` to the variable that is mapped to `x` in `s` if any, or `None`.

```
class (Substitution s, MaybeType sx) ⇒ MapsTo s x sx | s x → sx

instance MapsTo SubstEmpty x None
instance Substitution s ⇒ MapsTo (SubstCons x sx s) x (Some sx)
instance (Substitution s, MapsTo s x sx) ⇒ MapsTo (SubstCons x' sx' s) x sx
```

Here, finding an instance of `MapsTo s x sx` amounts to finding `sx` such that the substitution `s` maps `x` to `sx`. Note the use of functional dependency in the type class definition (the `| s x → sx` annotation in the definition above) to ensure that `MapsTo` is actually a "function" from substitution and type to `MaybeType`. Functional dependencies were proposed by Jones [8] as a mechanism to more precisely control type inference in Haskell. An expression `c e | c → e` means that fixing the type `c` should fix the type `e`.

2.4. Statically Computing Least General Types

We now show how to encode the anti-unification algorithm λ described in Section 2.2 at the level of types, exploiting the type class mechanism of Haskell.

The type class `LeastGen` is defined as a particular case of the more general type class `LeastGen'`, shown in Figure 1. This class is defined in line 1 and is parameterized by types a, b, c, σ_{in} and σ_{out}. σ_{in} and σ_{out} denote substitutions encoded at the type level as a list of mappings from type variables to *pairs* of types. We use pairs of types in substitutions because we have to simultaneously compute substitutions from c to a and from c to b. To be concise, lines $5-8$ presents a single definition parametrized by the type constructor arity but in practice, a different instance declaration has to be added for each type constructor arity.

Proposition 1. If `LeastGen'` $a\ b\ c\ \sigma_{in}\ \sigma_{out}$ holds, then the substitution σ_{out} extends σ_{in} and $\sigma_{out}c = (a, b)$.

Proof. By induction on the type representation of a and b.

A type can either be a type variable, represented as `TVar a`, or an n-ary type constructor `T` applied to n type arguments[3]. The rule to be applied depends on whether the type constructors of a and b are the same or not.

(*i*) If the constructors are the same, the rule defined in lines 5-8 computes `(T` $c_1\ \dots\ c_n$`)` using the induction hypothesis that $\sigma_i c_i = (a_i, b_i)$, for $i = 1 \dots n$. The component-wise application of constraints is done from left to right, starting from substitution σ_0 and extending it to the resulting substitution σ_n. The type equality constraint `(T` c_1 `...)` $\sim c$ checks that `c` is unifiable with `(T` c_1 `...)` and, if so, unifies them. Then, we can check that $\sigma_n c = (a, b)$.

(*ii*) If the type constructors are not the same the only possible generalization is a type variable. In the rule defined in lines 12-16 the goal is to extend σ_{in} with the mapping $c \mapsto (a, b)$ such that $\sigma_{out}c = (a, b)$, while preserving the injectivity of the substitution (see next proposition). ☐

Proposition 2. If σ_{in} is an injective function, and `LeastGen'` $a\ b\ c\ \sigma_{in}\ \sigma_{out}$ holds, then σ_{out} is an injective function.

Proof. By construction `LeastGen'` introduces a binding from a fresh type variable to (a, b), in the rule defined in lines 12-16, only if there is no type variable already mapping to (a, b)—in which case σ_{in} is not modified.

To do this, we first check that c is actually a type variable `(TVar` c`)` by checking its representation using `Analyze`. Then in relation `MapsTo` we bind c' to the (possibly inexistent) type variable that maps to (a, b) in σ_{in}. In case there is no such mapping c' is `None`.

Finally, relation `VarCase` binds σ_{out} to σ_{in} extended with $\{c \mapsto (a, b)\}$ in case c' is `None`, otherwise $\sigma_{out} = \sigma_{in}$. It then unifies c with c'. In all cases c is bound to the variable that maps to (a, b) in σ_{out}, because it was either unified in rule `MapsTo` or in rule `VarCase`.

The hypothesis that σ_{in} is injective ensures that any preexisting mapping is unique. ☐

Proposition 3. If σ_{in} is an injective function, and `LeastGen'` $a\ b\ c\ \sigma_{in}\ \sigma_{out}$ holds, then c is the least general type of a and b.

Proof. By induction on the type representation of a and b.

(*i*) If the type constructors are different the only generalization possible is a type variable c.

(*ii*) If the type constructors are the same, then $a = Ta_1 \dots a_n$ and $b = Tb_1 \dots b_n$. By Proposition 1 , $c = Tc_1 \dots c_n$ generalizes a and b with the substitution σ_{out}. By induction hypothesis c_i is the least general type of (a_i, b_i).

Now consider a type d that also generalizes a and b, *i.e.* $a \preceq d$ and $b \preceq d$, with associated substitution α. We prove c is less general than d by constructing a substitution τ such that $\tau d = c$.

Again, there are two cases, either d is a type variable, in which case we set $\tau = \{d \mapsto c\}$, or it has the same outermost type constructor, *i.e.* $d = Td_1 \dots d_n$. Thus $a_i \preceq d_i$ and $b_i \preceq d_i$; and since c_i is the least general type of a_i and b_i, there exists a substitution τ_i such that $\tau_i d_i = c_i$, for $i = 1 \dots n$.

Now consider a type variable $x \in dom(\tau_i) \cap dom(\tau_j)$. By definition of α, we know that $\sigma_{out}(\tau_i(x)) = \alpha(x)$ and $\sigma_{out}(\tau_j(x)) = \alpha(x)$. Because σ_{out} is injective (by Proposition 2), we deduce that $\tau_i(x) = \tau_j(x)$ so there are no conflicting mappings between τ_i and τ_j, for any i and j. Thus we can define $\tau = \bigcup \tau_i$ and check that $\tau d = c$. ☐

Definition 4 (`LeastGen` type class)**.** To compute the least general type `c` for `a` and `b`, we define:

`LeastGen a b c` $\triangleq$ `LeastGen' a b c SubstEmpty` σ_{out}, where `SubstEmpty` is the empty substitution and σ_{out} is the resulting substitution.

[3] We use the `Analyze` type class from `PolyTypeable` to statically distinguish type structure. An instance `Analyze a r` holds if `r` is the type representation of `a`. `TVar` is a simple type constructor used to explicitly tag type variables at the type level. For simplicity we omit the rules for analyzing type representations.

2.5. Playing Around with GHC Extensions

Up to now, we have assumed that the Haskell type class resolution mechanism is choosing the proper type class instances (or clauses) at each step of the algorithm. But the situation is more complicated than that. First, our type class instances are too complicated for Haskell to be able to decide if type class resolution will ever terminate. For that reason, we need the `UndecidableInstances` GHC extension to force Haskell to accept such instances.

Moreover, we have different instances that can be applied to the same resolution, which is called *overlapping instances*. For instance the default case defined in lines 12-16 of Figure 1 is overlapping with every other instances. For that specific case, we just need to add the `OverlappingInstances` GHC extension because all others instances are more specific than the default case, so we can tell Haskell to always prefer the most specific instances.

Ideally, we would like that all overlapping instance problems can be solved by giving the priority to the most specific instances. Unfortunately, the construction of substitution using the type class `MapsTo` requires two instances:

```
instance Substitution s ⇒ MapsTo (SubstCons x sx s) x (Some sx)
instance (Substitution s, MapsTo s x sx) ⇒ MapsTo (SubstCons x' sx' s) x sx
```

depending if the head of the substitution deals with the same `x`. But during the algorithm, we encounter situations like:

```
MapsTo (SubstCons (a, b) c SubstEmpty) (a, b') d
```

where unifying `b` with `b'` allows to use the first instance, and not unifying allows to use the second instance. In that specific situation, no instance is more general than the other, so the `OverlappingInstances` extension is not sufficient to tell Haskell which instance to use. Here, we have to use the (apparently dangerous) `IncoherentInstances` extension to force Haskell to pick up the instance that was (syntactically) declared last. This is correct because in the generalization process, two variables that are distinct must not be unified.

2.6. Taking Type Class Constraints into Account

The proposed anti-unification type class only works for plain types—that is types without constraints. Taking type class constraints into account in the computation of the least generalization would be very useful in practice, but it requires to solve two main issues: (i) what is the right definition of least generalization in presence of type class constraints, (ii) how to deal with type class constraints in the definition of `LeastGen`. The second issue seems the most serious as type class constraints are not really part of the Haskell type system in the sense that it is not possible to reify constraints, at least with the `Polytypeable` library. Maybe this functionality will be provided by a future GHC extension.

3. A Typed Functional Embedding of First-Class Aspects

In this section we present an application of anti-unification at the level of types to define a type-safe embedding of aspects in Haskell. We start with a brief overview of aspect-oriented programming and its applications. After that we exemplify our approach to purely functional aspects, to then describe in detail our embedded model of AOP in Haskell. This section only describes the aspect-oriented model, we discuss type safety – achieved with anti-unification – in Section 4.

The code presented below is (a simplified) part of a larger project called haskellaop, which provides aspects in Haskell. The project can be found at http://pleiad.cl/haskellaop.

3.1. An Overview of Aspect-Oriented Programming

Aspect-oriented programming (AOP) is a programming paradigm originally proposed by Kiczales *et al.* [10] to modularize *crosscutting-concerns*. A concern is crosscutting if it can not be modularized by the dominant decomposition mechanism of a given language (*e.g.* functions, procedures or objects), and thus such concerns are scattered among many modules of the software system. As a solution, AOP provides *aspects* as modular units that encompass crosscutting behavior. Typical examples of crosscutting concerns include dynamic analysis aspects [21], error handling [3], and persistence [18].

We focus on the pointcut-advice model for AOP [12], which is used by mainstream AOP languages such as AspectJ [9], and research languages like AspectScheme [6] and AspectML [4]. In the pointcut-advice model, *join points* represent events during program execution (function call, variable assignment, etc.), which are identified by predicates called *pointcuts*. *Advice* is the definition of crosscutting behavior associated with join points. An *aspect* is a modular entity composed of pointcut-advice pairs, whose semantics are such that whenever a pointcut matches, its corresponding advice executes. This semantics is obtained by using a *weaving* process that inserts the crosscutting behavior in the right parts of the original program. Such a mechanism is typically integrated in an existing programming language by modifying the language processor, may it be the compiler (either directly or through macros), or the virtual machine.

In a statically typed language, introducing pointcuts and advices also means extending the type system, if type soundness is to be preserved. For instance, AspectML [4] is based on a specific type system in order to safely apply advice. AspectJ [9] does not substantially extend the type system of Java and suffers from soundness issues. StrongAspectJ [5] addresses these issues with an extended type system. In both cases, proving type soundness is rather involved because a whole new type system has to be dealt with. In contrast, we provide a lightweight approach to embed aspects in Haskell, in a type-safe manner.

Note that although typical applications of aspects involve stateful computations, in this paper we only consider pure aspects. The embedding of aspects in Haskell can be extended to deal with effects, and our full-fledged implementation actually supports them. However, effects are not relevant for illustrating the application of anti-unification.

3.2. Purely Functional Aspects

A premise for aspect-oriented programming in functional languages is that function applications are subject to aspect weaving. We introduce the term *open application* to refer to a function application that generates a join point, and consequently, can be woven. In this paper, open applications are realized explicitly using the `#` operator: `f # 2` is the same as `f 2`, except that the application generates a join point that is subject to aspect weaving.

As a basic example, consider the following:

```
 monadic version of sqrt and chr
sqrtM n = return (sqrt n)
chrM  n = return (chr n)

 advice:
ensurePos proceed n = proceed (abs n)

 using an aspect:
program n = do deploy (aspect (pcOr (pcCall sqrt) (pcCall chr)) ensurePos)
               x ← sqrtM # n
               y ← chrM # n
               return (x,y)
```

The advice `ensurePos` enforces that the argument of a function application is a positive number, by replacing the original argument with its absolute value. We then deploy an aspect that reacts to applications of either `sqrtM` or `chrM` (`chrM` yields the unicode character indexed by a given integer, which should be positive). This is specified using the pointcut `(pcOr (pcCall sqrtM) (pcCall chrM))`. Evaluating `program -4` results in `sqrtM` and `chrM` to be eventually applied with argument `4`. As can be seen, aspects are created with `aspect` and deployed with `deploy`.

Observe that `ensurePos` can use the `abs` operation on `n` because all the advised functions have a numeric argument. However, since the return types of these functions are different, `ensurePos` can only use the most common type pattern between them, which in this case is a fresh type variable. It is therefore crucial to be able to perform anti-unification at the type level, to guarantee type safety without imposing severe restrictions on the available pointcuts.

It may appear contradictory that our example shows monadic code, after we stated that we are not modeling computational effects (which in Haskell is done using monads). The reason is that we maintain the list of currently deployed aspects using a state monad, described below in Section 3.4.

Our introduction of AOP simply relies on defining aspects (pointcuts, advices), the underlying aspect environment together with the operations to deploy and undeploy aspects, and open function application. The remainder of this section briefly presents these elements, and the following section concentrates on the main challenge: properly typing pointcuts and ensuring type soundness of pointcut/advice bindings.

3.3. Join Point Model

We now describe the elements of the pointcut-advice model: join points, pointcuts, and advices.

Join points. Join points are function applications. A join point `JP` contains a function of type `a → b`, and an argument of type `a`.

```
data JP a b = (PolyTypeable (a → b)) ⇒ JP (a → b) a
```

In addition, we require functions to be `PolyTypeable` because, by default, the type of a function is not available at runtime. This limitation prohibits to define generic pointcuts that match a specific type signature. To overcome this limitation we use the `PolyTypeable` library, which adds introspection capabilities to monomorphic and polymorphic functions – so it is even possible to advise polymorphic functions [4].

Pointcuts. A pointcut is a predicate on the current join point. It is used to identify join points of interests. A pointcut simply returns a boolean to indicate whether it matches the given join point.

```
data PC a b = PC (∀ a' b'. (JP a' b' → Bool))
```

A pointcut is a function of type `∀ a' b'. (JP a' b' → Bool)`. The `∀` declaration quantifies on type variables `a'` and `b'` (using rank-2 types) because a pointcut should be able to match against any join point, regardless of the specific types involved (we come back to this in Section 4.1).

As the intermediary between a join point and an advice is the pointcut, whose proper typing is therefore crucial. The type of a pointcut as a predicate over join points does not convey any information about the types of join points it matches. To keep this information, we use *phantom type variables* `a` and `b` in the definition of `PC`. A phantom type variable is a type variable that is not used on the right hand-side of the data type definition. The use of phantom type variables to type embedded languages was first introduced by Leijen and Meijer to type an embedding of SQL in Haskell [11]; it makes it possible to "tag" extra type information on data. In our context, we use it to add the information about the type of the join points matched by a pointcut: `PC a b` means that a pointcut can match join points of type `a → b`. We call this type the *matched type* of the pointcut. Pointcut designators are in charge of specifying the matched type of the pointcuts they produce.

We provide two basic pointcut designators, `pcCall` and `pcType`, as well as logical pointcut combinators, `pcOr`, `pcAnd`, and `pcNot`.

[4] From now on, we omit the type constraints related to `PolyTypeable` (the `PolyTypeable` constraint on a type is required each time the type has to be inspected dynamically; exact occurrences of this constraint can be found in the implementation).

```
pcType f = let t = polyTypeOf f in PC (_type t)
           where _type t = \jp → compareType t jp

pcCall f = let t = polyTypeOf f in PC (_call f t)
           where _call f t = \jp → compareFun f jp && compareType t jp
```

`pcType f` matches all calls to functions that have a type compatible with `f` (see Section 4.1 for a detailled definition) while `pcCall f` matches all calls to `f`. In both cases, `f` is constrained to allow using the `PolyTypeable` introspection mechanism, which provides the `polyTypeOf` function to obtain the type representation of a value. This is used to compare types with `compareType`.

To implement `pcCall` we require a notion of function equality[5]. This is used in `compareFun` to compare the function in the join point to the given function. Note that we also need to perform a type comparison, using `compareType`. This is because a polymorphic function whose type variables are instantiated in one way is equal to the same function but with type variables instantiated in some other way (*e.g.* `id :: Int → Int` is equal to `id :: Float → Float`).

Advice. An advice is a function that executes in place of a join point matched by a pointcut. This replacement is similar to open recursion in EffectiveAdvice [13]. An advice receives a function (known as the `proceed` function) and returns a new function of the same type (which may or may not apply the original `proceed` function internally). We introduce a type alias for advice:

```
type Advice a b = (a → b) → a → b
```

For instance, the type `Advice Int Int` is a synonym for the type `(Int → Int) → Int → Int`. For a given advice of type `Advice a b`, we call `a → b` the *advised type* of the advice.

Aspect. An aspect is a first-class value binding together a pointcut and an advice. Supporting first-class aspects is important: it makes it possible to support aspect factories, separate creation and deployment/undeployment of aspects, exporting opaque, self-contained aspects as single units, etc. We introduce a data definition for aspects:

```
data Aspect a b c d = Aspect (PC a b) (Advice c d)
```

We defer the detailed definition of `Aspect` with its type class constraints to Section 4.2, when we address the issue of safe pointcut/advice binding.

3.4. Aspect Weaving

The list of aspects that are deployed at a given point in time is known as the *aspect environment*. To be able to define an heterogenous list of aspects, we use an existentially-quantified data `EAspect` that hides the type parameters of `Aspect`[6]:

```
data EAspect = ∀ a b c d. EAspect (Aspect a b c d)
type AspectEnv = [EAspect]
```

This environment can be either fixed initially and used globally [12], as in AspectJ, or it can be dynamic, as in AspectScheme [6]. Different scoping strategies are possible when dealing with dynamic deployment [20].

Here, we propose to embed the aspect environment inside a monad similar to the state monad

```
data AO a = AO {run :: AspectEnv → (a, AspectEnv)}
```

We use a **`data`** declaration to define the type `AO`. This type wraps a `run` function, which takes an initial aspect environment and returns a value of type `a`, and a potentially modified aspect environment. The monadic `bind` and `return` functions of the `AO` monad are the same as in the state monad.

[5] For this notion of function equality, we use the `StableNames` API, which relies on pointer comparison.

[6] Since existential quantification requires type parameters to be free of type class constraints, aspects with ad-hoc polymorphism have to be instantiated before deployment to statically solve each remaining type class constraint.

We now define the functions for dynamic deployment, which simply add and remove an aspect from the aspect environment (note the use of `$` to avoid extra parentheses):

```
deploy, undeploy :: EAspect → AO ()
deploy asp   = AO $ \asps → ((), asp:asps)
undeploy asp = AO $ \asps → ((), deleteAsp asp asps)
```

To extract the value from an `AO` computation we define the `runAO` function, with type `AO a → a` (similar to `evalState` in the state monad), that runs a computation in an empty initial aspect environment. For instance, in the example of the `sqrt` function, we can define a `client` as follows:

```
client n = runAO (program n)
```

Weaving. The function to use at a given point is produced by the `weave` function, defined below:

```
weave :: (a → AO b) → AspectEnv → JP a b → (a → AO b)
weave f [] jp = f
weave f env@(asp:asps) jp =
    case asp of EAspect (Aspect pc adv) →
      let (match,_) = apply_pc pc jp env
      in weave (if match
                   then apply_adv adv f
                   else f)
               asps jp
```

The `weave` function is defined recursively on the aspect environment. For each aspect, it applies the pointcut to the join point. It then uses either the partial application of the advice to `f` if the pointcut matches, or `f` otherwise, to keep on weaving on the rest of the aspect list. `apply_pc` checks whether the pointcut matches the join point, and returns a pair `(match, aenv')` with a boolean and a potentially new aspect environment – which we discard as a design choice, given that in AOP languages it is not common that a pointcut can persistently deploy aspects. This definition of weaving is a direct adaptation of AspectScheme's weaving function [6]. Then, the open application can be defined as

```
(#) :: (a → AO b) → a → AO b
f # a = AO $ \asps → run (weave f asps (newjp f a) a) asps
```

where `newjp` is the constructor of join points.

Applying Advice. As we have seen, the aspect environment has type `AspectEnv m`, meaning that the type of the advice function is hidden. Therefore, advice application requires *coercing* the advice to the proper type in order to apply it to the function of the join point:

```
apply_adv :: Advice a b → t → t
apply_adv adv f = (unsafeCoerce adv) f
```

The operation `unsafeCoerce` of Haskell is (unsurprisingly) unsafe and can yield to segmentation faults or arbitrary results. To recover safety, we could insert a runtime type check with `compareType` just before the coercion. We instead make aspects type safe such that we can prove that the use of `unsafeCoerce` in `apply_adv` is *always* safe. The following section describes how we achieve type soundness of aspects by relying on anti-unification.

4. Type-Safe Aspects with Anti-Unification

Ensuring type soundness in the presence of aspects consists in ensuring that an advice is always applied at a join point of the proper type. Note that by "the type of the join point", we refer to the type of the function being applied at the considered join point.

Our type-safe embedding requires anti-unification at three different places: (i) when computing the matched type of the disjunction of two pointcuts (using the `pcOr` poincut combinator), (ii) when composing user-defined pointcuts, and (iii) when binding a pointcut and an advice. This section describes precisely how pointcuts, advices and aspects are typed and shows type soundness: no advice application can go wrong.

4.1. Typing Pointcuts

Least General Types. Because a pointcut potentially matches many join points of different types, the associated type must be a *more general type*. For instance, consider a pointcut that matches applications of functions of type `Int → Int` and `Float → Int`. Its matched type is the parametric type `a → Int`. Note that this is in fact the *least general type* of both types.[7] Another more general candidate is `a → b`, but the least general type conveys more precise information.

As a concrete example, below is the type signature of the `pcCall` pointcut designator:

```
pcCall :: (a → b) → PC a b
```

Note the second argument to the `PC` type constructor: it specifies that a call pointcut matches applications of a function of type `a → b`, which is precisely the type of the function passed to `pcCall`.

Comparing Types. The type signature of the `pcType` pointcut designator is the same as that of `pcCall`:

```
pcType :: (a → b) → PC a b
```

However, suppose that `f` is a function of type `Int → a`. We want the pointcut `(pcType f)` to match applications of functions of more specific types, such as `Int → Int`. This means that `compareType` actually checks that the matched type of the pointcut is *more general* than the type of the join point.

Logical Combinators. We use type constraints in order to properly specify the matched type of logical combinators. The intersection of two pointcuts matches join points that are most precisely described by the *principal unifier* of both matched types. Since Haskell supports this unification when the same type variable is used, we can simply define `pcAnd` as follows:

```
pcAnd :: PC a b → PC a b → PC a b
```

For instance, a control flow pointcut matches any type of join point, so its matched type is `a → b`. Consequently, if `f` is of type `Int → a`, the matched type of `pcAnd (pcCall f) (cflow g)` is `Int → a`.

Dually, the union of two pointcuts relies on anti-unification:

```
pcOr :: (LeastGen (a → b) (c → d) (e → f)) ⇒
        PC a b → PC c d → PC e f
```

For instance, if `f` is of type `Int → a` and `g` is of type `Int → Float`, the matched type of `pcOr (pcCall f) (pcCall g)` is `Int → a`.

The negation of a pointcut can match join points of any type because no assumption can be made on the matched join points:

```
pcNot :: PC a b → PC a' b'
```

Observe that the type of `pcNot` is quite restrictive. In fact, the advice of any aspect with a single `pcNot` pointcut must be completely generic. The matched type of `pcNot` can be made more specific using `pcAnd` to combine it with other pointcuts with more specific types.

User-defined Pointcut Designators. The set of pointcut designators in our language is open. User-defined pointcut designators are however responsible for properly specifying their matched types. If the matched type is incorrect or too specific, soundness is lost.

A pointcut cannot make any type assumption about the type of the join point it receives as argument. The reason for this is again the homogeneity of the aspect environment: when deploying an aspect, the type of its pointcut is hidden. At runtime, then, a pointcut is expected to be applicable to any join point. The general approach to make a pointcut safe is therefore to perform a runtime type check, as was illustrated in the definition of `pcCall` and `pcType` in Section 3.3. However, certain pointcuts are meant to be conjuncted with others pointcuts

[7]The term *most specific generalization* is also valid, but we stick here to Plotkin's original terminology [17].

that will first apply a sufficient type condition.

In order to support the definition of pointcuts that *require* join points to be of a given type, we provide the `RequirePC` type:

```
data RequirePC a b = RequirePC (∀ a' b'. (JP a' b' → Bool))
```

The definition of `RequirePC` is similar to that of `PC`, with two important differences. First, the matched type of a `RequirePC` is interpreted as a type *requirement*. Second, a `RequirePC` is not a valid stand-alone pointcut: it has to be combined with a standard `PC` that enforces the proper type upfront. To safely achieve this, we overload `pcAnd`[8]:

```
pcAnd :: (LessGen (a → b) (c → d)) ⇒ PC a b → RequirePC c d → PC a b
```

`pcAnd` yields a standard `PC` pointcut and checks that the matched type of the `PC` pointcut is *less general* than the type expected by the `RequirePC` pointcut. This is expressed using the constraint `LessGen`, whose definition relies directly on `LeastGen`:

```
LessGen a b ≜ LeastGen a b b
```

To illustrate, let us define a poincut designator `pcArgGT` for specifying pointcuts that match when the argument at the join point is greater than a given `n` (of type `a` instance of the `Ord` type class):

```
pcArgGT :: (Ord a) ⇒ a → RequirePC a b
pcArgGT n = RequirePC $ (\jp → unsafeCoerce (getJpArg jp) >= n)
```

The use of `unsafeCoerce` to coerce the join point argument to the type `a` forces us to declare the `Ord` constraint on `a` when typing the returned pointcut as `RequirePC a b` (with a fresh type variable `b`). To get a proper pointcut, we use `pcAnd`, for instance to match all calls to `sqrt` where the argument is greater than 10:

```
pcCall sqrt `pcAnd` pcArgGT 10
```

The `pcAnd` combinator guarantees that a `pcArgGT` pointcut is always applied to a join point with an argument that is indeed of a proper type: no runtime type check is necessary within `pcArgGT`, because the coercion is always safe.

4.2. Typing Aspects

The typing issue we have to address consists in ensuring that a pointcut/advice binding is type safe, so that the advice application does not fail. A first idea to ensure that the pointcut/advice binding is type safe is to require the matched type of the pointcut and the advised type of the advice to be the same (or rather, unifiable):

wrong!

```
data Aspect a b = Aspect (PC a b) (Advice a b)
```

This approach can however yield unexpected behavior. Consider the following example:

```
idM :: a → AO a
idM a = return a

adv :: Advice (Char → AO Char)
adv proceed c = proceed (toUpper c)

program = do deploy (aspect (pcCall id) adv)
             x <- idM # 'a'
             y <- idM # [True,False,True]
             return (x,y)
```

The matched type of the pointcut `pcCall idM` is `a → AO a`. With the above definition of `Aspect`, `program` passes the typechecker because it is possible to unify `a` and `Char` to `Char`. However, when evaluated, the behavior of `program` is undefined because the advice is unsafely applied with an argument of type `[Bool]`, for which `toUpper` is undefined.

The problem is that during typechecking, the matched type of the pointcut and the advised type of the

[8]The constraint is different from the previous constraint on `pcAnd`. This is possible thanks to the recent `ConstraintKinds` extension of `ghc`.

advice can be unified. Because unification is symmetric, this succeeds even if the advised type is more specific than the matched type. Again, we use the type class `LessGen` to ensure that the matched type is less general than the advice type:

```
data Aspect a b c d = LessGen (a → b) (c → d) ⇒  Aspect (PC a b) (Advice c d)
```

This constraint ensures that pointcut/advice bindings are type safe: the coercion performed in `apply_adv` always succeeds. We formally prove this in the following section.

4.3. Pointcut Safety

We now establish the safety of pointcuts with relation to join points.

Definition 5 (Pointcut match)**.** We define the relation `matches(pc,jp)`, which holds iff applying pointcut `pc` to join point `jp` in the context of a monad `m` yields a computation `m True`.

Now we prove that the matched type of a given pointcut is more general than the join points matched by that pointcut.

Proposition 4. Given a join point term `jp` and a pointcut term `pc`, and type environment Γ,
if $\Gamma \vdash$ `pc` : `PC a b` $\quad \Gamma \vdash$ `jp` : `JP a' b'` $\quad \Gamma \vdash$ `matches(pc, jp)`
then `a'` $\rightarrow$ `b'` $\preceq$ `a` $\rightarrow$ `b`.

Proof. By induction on the matched type of the pointcut.

- Case `pcCall`: By construction the matched type of a `pcCall f` pointcut is the type of `f`. Such a pointcut matches a join point with function `g` if and only if: `f` is equal to `g`, and the type of `f` is less general than the type of `g`. (On both `pcCall` and `pcType` this type comparison is performed by `compareType` on the type representations of its arguments.)
- Case `pcType`: By construction the matched type of a `pcType f` pointcut is the type of `f`. Such a pointcut only matches a join point with function `g` whose type is less general than the matched type.
- Case `pcAnd` on `PC PC`: Consider `pc1 ‘pcAnd‘ pc2`. The matched type of the combined pointcut is the *principal unifier* of the matched types of the arguments—which represents the intersection of the two sets of joinpoints. The property holds by induction hypothesis on `pc1` and `pc2`.
- Case `pcAnd` on `PC RequirePC`: Consider `pc1 ‘pcAnd‘ pc2`. The matched type of the combined pointcut is the type of `pc1` and it is checked that the type required by `pc2` is *more general* so the application of `pc2` will not yield an error. The property holds by induction hypothesis on `pc1`.
- Case `pcOr`: Consider `pc1 ‘pcOr‘ pc2`. The matched type of the combined pointcut is the *least general type* of the matched types of the argument, computed by the `LeastGen` constraint—which represents the union of the two sets of joinpoints. The property holds by induction hypothesis on `pc1` and `pc2`.
- Case `pcNot`: The matched type of a pointcut constructed with `pcNot` is a fresh type variable, which by definition is more general than the type of any join point.
- User-defined pointcuts must maintain this property, otherwise safety is lost.

□

4.4. Advice Type Safety

If an aspect is well-typed, the advice is more general than the matched type of the pointcut:

Proposition 5. Given a pointcut term `pc`, an advice term `adv`, and a type environment Γ,
if $\Gamma \vdash$ `pc` : `PC a b` $\quad \Gamma \vdash$ `adv` : `Advice c d` $\quad \Gamma \vdash$ `(aspect pc adv)` : `Aspect a b c d`
then `a` $\rightarrow$ `b` $\preceq$ `c` $\rightarrow$ `d`.

Proof. Using the definition of `Aspect` (Section 4.2) and because $\Gamma \vdash$ `(aspect pc adv)` : `Aspect a b c d`, we know that the constraint `LessGen` is satisfied, so by Definitions 4 and 5, and Proposition 1, `a` $\rightarrow$ `b` $\preceq$ `c` $\rightarrow$ `d`. □

4.5. Safe Aspects

We now show that if an aspect is well-typed, the advice is more general than the advised join point:

Theorem 1 (Safe Aspects)**.** *Given the terms* `jp`, `pc` *and* `adv` *representing a join point, a pointcut and an advice respectively, given a type environment* Γ*,*

if	$\Gamma \vdash$ `pc` : `PC a b`	$\Gamma \vdash$ `adv` : `Advice c d`	$\Gamma \vdash$ `(aspect pc adv)` : `Aspect a b c d`
and	$\Gamma \vdash$ `jp` : `JP a' b'`	$\Gamma \vdash$ `matches(pc, jp)`	
then	`a'` $\rightarrow$ `b'` $\preceq$ `c` $\rightarrow$ `d`.		

Proof. By Proposition 4 and 5 and the transitivity of $\preceq$. □

Corollary 1 (Safe Advice Application)**.** The coercion of the advice in `apply_adv` is safe.

Proof. Recall `apply_adv` (Section 3.4):

```
apply_adv :: Advice a b → t → t
apply_adv adv f = (unsafeCoerce adv) f
```

By construction, `apply_adv` is used only with a function `f` that comes from a join point that is matched by a pointcut associated to `adv`. Using Theorem 1, we know that the join point has type `JP a' b'` and that `a'` $\rightarrow$ `b'` $\preceq$ `a` $\rightarrow$ `b`. We note σ the associated substitution. Then, by compatibility of substitutions with the typing judgement [16], we deduce $\sigma\Gamma \vdash \sigma$`adv` : `Advice a' b'`. Therefore `(unsafeCoerce adv)` corresponds exactly to σ`adv`, and is safe. □

5. Conclusion

To conclude, we believe that the anti-unification algorithm using type classes is a good illustration of the potential benefit of the Haskell type class system to encode specific type-level algorithms. The resulting type classes can then be used to type a language embedding that requires typing notions that are not already present in Haskell type system. We are interested in investigating the use of the anti-unification type class in other contexts.

Acknowledgements This work was supported by the INRIA Associated team RAPIDS.

References

[1] M. Alpuente, S. Escobar, J. Meseguer, and P. Ojeda. Order-sorted generalization. *Electron. Notes Theor. Comput. Sci.*, 246:27–38, Aug. 2009.

[2] *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*, Brussels, Belgium, Apr. 2008. ACM Press.

[3] R. Coelho, A. Rashid, A. Garcia, N. Cacho, U. Kulesza, A. Staa, and C. Lucena. Assessing the impact of aspects on exception flows: An exploratory study. In J. Vitek, editor, *Proceedings of the 22nd European Conference on Object-oriented Programming (ECOOP 2008)*, number 5142 in Lecture Notes in Computer Science, pages 207–234, Paphos, Cyprus, july 2008. Springer-Verlag.

[4] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems*, 30(3):Article No. 14, May 2008.

[5] B. De Fraine, M. Südholt, and V. Jonckers. StrongAspectJ: flexible and safe pointcut/advice bindings. In AOSD 2008 [2], pages 60–71.

[6] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, Dec. 2006.

[7] G. Huet. *Résolution d'équations dans les languages d'ordre 1,2,...,ω*. PhD thesis, Université de Paris VII, 1976.

[8] M. P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP '00, pages 230–244, London, UK, UK, 2000. Springer-Verlag.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.

[11] D. Leijen and E. Meijer. Domain specific embedded compilers. In T. Ball, editor, *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages*, pages 109–122, 1999.

[12] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.

[13] B. C. d. S. Oliveira, T. Schrijvers, and W. R. Cook. EffectiveAdvice: discplined advice with explicit effects. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, pages 109–120, Rennes and Saint Malo, France, Mar. 2010. ACM Press.

[14] B. Østvold. A functional reconstruction of anti-unification. Technical Report DART/04/04, Norwegian Computing Center, 2004.

[15] F. Pfenning. Unification and anti-unification in the calculus of constructions. In *Logic in Computer Science, 1991. LICS'91., Proceedings of Sixth Annual IEEE Symposium on*, pages 74–85. IEEE, 1991.

[16] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[17] G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

[18] A. Rashid and R. Chitchyan. Persistence as an aspect. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, AOSD '03, pages 120–129, New York, NY, USA, 2003. ACM.

[19] J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5:135–151, 1970.

[20] É. Tanter. Expressive scoping of dynamically-deployed aspects. In AOSD 2008 [2], pages 168–179.

[21] É. Tanter, P. Moret, W. Binder, and D. Ansaloni. Composition of dynamic analysis aspects. In *Proceedings of the 9th ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE 2010)*, pages 113–122, Eindhoven, The Netherlands, Oct. 2010. ACM Press.

[22] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM.