Automating Gradual Typing

Timothy Jones
Victoria University of Wellington
tim@ecs.vuw.ac.nz

1. Introduction

Gradual typing [4] has become a standard mechanism for both progressively adding types to dynamically typed languages and liberalising existing type systems, and recent developments have produced the Gradualizer [1] and Abstracting Gradual Typing (AGT) [2] for systematically lifting a typed language to a corresponding gradual language. AGT applies the principles of abstract interpretation to a language's underlying predicates and functions: we have used this technique to build a library in the dependently typed language Agda [3] for automatically generating different type systems over the same language of terms. Our mechanisation takes an abstract description of a type system indexed by a functor and then generates a variety of different type systems, including gradual typing.

2. Mechanisation

As a dependently typed language, Agda allows the definition of complex propositions about terms, including formal languages. If we consider the Simply Typed Functional Language from AGT, the syntax of its types would be encoded as:

```
data Type : Set where
  Int : Type
  Bool : Type
  _→_ : (T₁ T₂ : Type) Type
```

Then the syntax of terms such that each term includes a proof of its own well-typedness, ensuring only well-formed terms can ever be constructed (note that we omit implicit bindings in the listed Agda):

```
data Term (Γ : Type n) : Type → Set where
  int : ℤ → Term Γ Int
  …
```

Applying AGT to this mechanisation to generate a corresponding gradual language, we reproduce all of these definitions to work under GType, which extends Type with an additional member to represent the unknown type (?) and the equality relation (≡) lifted into its consistent form (≅). In order to lift predicates like equality into their consistent form, AGT defines a *concretisation* function $\gamma : \mathsf{GType} \to \mathbb{P}\ \mathsf{Type}$ to describe which concrete types 'fit' into a gradual type. The resulting power sets may be infinite in size, and as such the actual set values cannot be computed: instead, the function must be encoded as a relation between gradual and concrete types.

```
data γ : GType → Type → Set where
  ? : γ ? T
  Int : γ Int Int
  Bool : γ Bool Bool
  _→_ : γ T̃₁ T₁ → γ T̃₂ T₂ → γ (T̃₁ → T̃₂) (T₁ → T₂)
```

Lifting a predicate from Types to GTypes using the concretisation function can then be defined exactly as in AGT.

```
data Lift² (P : BinRel Type) (T̃₁ T̃₂ : GType) : Set where
  raise : P T₁ T₂ → T₁ ∈ γ T̃₁ → T₂ ∈ γ T̃₂ → Lift² P T̃₁ T̃₂
```

For instance, the consistent equality relation ≅ is $\mathsf{Lift}^2\ \_\equiv\_$, and a proof that $(\mathsf{Int} \to ?) \cong (? \to \mathsf{Bool})$ first binds T_1 and T_2 to the type $(\mathsf{Int} \to \mathsf{Bool})$, then proves that T_1 and T_2 are equal and that the type is in γ of both of the given gradual types:

```
raise refl (Int → ?) (? → Bool)
```

Lift^2 can equally be applied to other relations between types, such as ≤ to produce consistent subtyping, and as in AGT the definition can be generalised to any predicate of finite arity.

Unlike AGT, we *cannot* implement such a lifting for partial functions. The operation to translate a function from Type to GType requires an *abstraction* function $\alpha : \mathbb{P}\ \mathsf{Type} \to \mathsf{GType}$ that reverses the concretisation of γ: this requires translating a potentially infinite input into a finite output, which cannot be expressed computationally. Fortunately, because the lifted functions are used in typing judgments, we can replicate the desired behaviour by expressing these functions as predicates of equality on their outcome:

```
data _≡—cod_ (T : Type) : Type → Set where
  refl : T ≡—cod (T′ → T)
```

For example, rather than asserting that the type of $(t_1 \cdot t_2)$ is $\mathsf{cod}\ T_1$, the outcome is bound to a new variable T_3, with $T_3 \equiv\!\!-\mathsf{cod}\ T_1$ as an extra premise. This predicate can then be lifted into its consistent form as normal. This does *not* preserve the optimality of α, as the outcome is not necessarily the most precise type; we are currently investigating the best mechanism to ensure an optimal lifting.

Applications of equality in the constructors of Term are then replaced with consistent equality, as in AGT. Other language definitions, potentially with different type relations, can be lifted using the same systematic approach. Currently, we do not address the lifting of reduction relations, though we plan to mechanise the AGT approach to this in the future.

3. Automation

So far we have discussed the application of the systematic lifting of a language in Agda, but can we go further than this? Agda is a programming language, so it would make sense that, rather than manually applying the lifting operation, we provide an abstraction in the form of a library that *automatically* transforms a typed language into its corresponding gradually typed language. This is achieved by defining a language with an abstracted type system.

First, we define a gradual type which is abstracted over the concrete components of the types, which is equivalent to the standard Maybe type constructor.

```
data Maybe A : Set where
  ? : Maybe A
  type : A → Maybe A
```

If we applied this directly to the previous Type, the result would be a type which is either unknown or entirely concrete, but we want to be able to embed the unknown type at any point of recursion in Type. As such, we require that an abstracted language's required

Type declaration be indexed by a unary type constructor. This gives us the *Abstractly Typed* Functional Language, with the types:

```
data Type (F : Set → Set) : Set where
  Int : Type F
  Bool : Type F
  _→_ : (T1 T2 : F (Type F)) → Type F
```

In the definition of the Terms of the language, we can require that F is a *unit functor*, having the standard functor mapping operation $\mathsf{lift} : (A \to B) \to F\,A \to F\,B$ and an extra operation from the applicative functor class $\mathsf{unit} : A \to F\,A$, along with proofs of the relevant functor and unit laws. The unit operation permits embedding concrete types into the given functor.

```
data Term (Γ : Vec (F (Type F)) n) : F (Type F) → Set where
  int : ℤ → Term Γ (unit Int)
  …
```

The abstract definition of the language allows the definition of both the full Type as id (Type id) (or just Type id) and the gradual GType as Maybe (Type Maybe), with corresponding functors, and unit defined as id and type respectively.

The missing component to interpret an abstract language as gradually typed is the concretisation function γ. The systematic approach of AGT relies on knowing the structure of the concrete types, but the general idea is that types with the same shape are related to each other, with points of recursion (which for the STFL is in the input and output of each function type) requiring corresponding recursive applications of γ. Type is indexed by a functor, and so may be mapped from one functor to another: it should be possible to perform a mapping between any two indices given a function between (Type specific) instances of the corresponding functors.

```
map : (F (Type G) → G (Type G)) → Type F → Type G
```

For the ATFL, this function maps Int to Int and Bool to Bool (as required by the associated laws), and for function types:

```
map f (T1 → T2) = f (lift (map f) T1) → f (lift (map f) T2)
```

The map function allows us to define γ in a way that is generic over any language which defines its types with this form of functor index. To achieve this, we observe that only the definition of Term requires that F be a unit functor. If we instead bind F to be an application of const to an Agda type (which is always a functor, but not necessarily a unit functor) in Type, then we can ignore the recursion of Type entirely and embed arbitrary values at the typical points of recursion instead. In this case, the values we wish to embed are pairs of gradual and concrete types, and accompanying proofs that γ holds for each pair.

By applying this observation, we get a proof structure for types essentially for free: Type (const (Σ (GType × Type) (uncurry γ))) (referred to below as Tγ). This structure can be broken apart into its gradual and concrete components with the map function.

```
data γ : GType → Type → Set where
  ? : γ ? T
  type : (T : Tγ) → γ (type (map proj11 T)) (map proj12 T)
```

The expected behaviour is guaranteed by the functor laws. The library can then use this generic form of γ to build Lift^2 and family for any participating abstract type system. The proofs are somewhat more painful as a result. (Int → ?) ≅ (? → Bool) now requires:

```
raise refl
  (type (((type Int , Int) , type Int) → ((? , Bool) , ?)))
  (type (((? , Int) , ?) → (type Bool , Bool) , type Bool))
```

Fortunately, Agda can infer the values of the gradual and concrete components, so ultimately the content is the same but with some extra boilerplate.

```
raise refl
  (type ((, type Int) → (, ?)))
  (type ((, ?) → (, type Bool)))
```

If the Terms of the language are defined alongside an abstract equality relation ≈, that relation can now be defined as ≡ for the language indexed by the unit functor id, or as ≅ for the language indexed by the unit functor Maybe. As long as the typing judgements follow the style required by AGT, this produces a gradual lifting operation for any type system.

The main caveat is that Agda has trouble reasoning about termination and positivity in such an abstracted environment, and it is difficult or outright impossible to communicate this information to Agda using the type system. At the very least, the most recent release of Agda has added a pragma to turn off positivity checks, so we can continue as described here and fall back on manually checking that the definitions are well-founded.

We make one final observation that although our goal was to build automated gradual type systems, there is no reason other type systems could not be built from other unit functors: for instance, the unit functor for const T immediately produces a dynamically typed language, and there are more bizarre but potentially useful opportunities such as List (terms have zero or more types) or × (types are tagged by some accompanying value). While the concretisation function γ given here is specific to the Maybe functor, savvy readers might have noticed that the structure of γ has a similar shape to the definition of Maybe, implying that each functor may have its own γ which can be generated automatically as well.

4. Conclusion

We have applied the principles of Abstracting Gradual Typing to a mechanisation of gradual type systems, and extended this into a library to build both concrete and gradual type systems from an abstract description. This required dealing with non-computational components of AGT, and finding the representation for types as indexed by functors in the abstract language description that allows for the definition of both the appropriate type values and their concretisation. In the future, we intend to apply the same principles to evaluation of typed terms as well.

Acknowledgments

Thanks to the anonymous reviewers for their comments. This work was supported in part by the Royal Society of New Zealand Marsden Fund.

References

[1] Matteo Cimini and Jeremy G. Siek. The Gradualizer: Gradual typing for free. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 443–455, 2016.

[2] Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 429–442, 2016.

[3] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[4] Jeremy Siek and Walid Taha. Gradual typing for function languages. In *Proc. Scheme and Functional Programming Workshop*, pages 81–92, 2006.