Synthesizing Well-Formed Programs using Datatype Generic Programming

CAS VAN DER REST, Utrecht University, The Netherlands

Advisor: Wouter Swierstra **Category:** Graduate (Master) **ACM student member number:** 2709516

INTRODUCTION

Test programs are invaluable when testing a compiler. However, synthesizing constrained test data such as well-formed programs is a difficult problem, which has been the subject of extensive research. For example: Claessen and Duregaard [2] generate constrained data using an iterative process in which they refine the search space in each step. Lampropoulos et al. define an abstraction over inductive relations in Coq, and describe how generators may be defined for these abstractions. They create an extension to QuickChick [4] implementing these ideas. Pałka et al. [9] generate well-typed lambda terms by reverse engineering a proof tree, given a goal type and context.

We take a novel approach by considering the more general problem of synthesizing values of *arbitrary indexed datatypes*, and generating values that inhabit indexed datatypes describing well-formedness. We use Dagand's universe of *Indexed Descriptions* [3] to represent indexed datatypes in a generic fashion. Given a description in this universe, we can derive a generator that produces values isomorphic to the type it represents. We have constructed a Haskell Library that implements this approach, where the user only needs to supply a description of a type describing well-formedness, and a conversion function that transforms generated values into the desired non-indexed datatype. We show how well-typed lambda terms can be generated using this technique. Additionally, we assert correctness of our work through a formalization in Agda. Both are available on Github [1].

THE UNIVERSE OF INDEXED DESCRIPTIONS

Indexed descriptions [3] are perhaps best viewed as an extension to the universe of regular types. The key modifications include:

- Types are described by a function $I \longrightarrow Description$, mapping indices to descriptions.
- Recursive positions store the index of the recursive subtree at that position.
- An extra constructor $\Sigma{:}(S{:}*) \longrightarrow (S \longrightarrow Description) \longrightarrow Description$ is added. The interpretation of $\Sigma\ S\ P$ is a *dependent pair* with type $((s{:}S), interpret \circ P)$.

We describe indexed descriptions using a GADT [7] with two type parameters: one denoting the associated non-indexed datatype, and the other the index type. The constructors for unit types, empty types, recursive positions, products and coproducts closely follow the definition of regular types, with the exception of *Var*, which stores an index value.

```
data IDesc (a :: *) (i :: *) where
  One   ::                               IDesc a i
  Empty ::                               IDesc a i
  Var   :: i                           → IDesc a i
  (:*:) :: IDesc a i → IDesc a i → IDesc a i
  (:+:) :: IDesc a i → IDesc a i → IDesc a i
```

We can then define a *type family* $Interpret\ (d :: IDesc\ a\ i) :: *$ that describes the semantics of a description. E is a datatype with no constructors representing the *empty type* ($\perp$).

Author's address: Cas van der Rest, c.r.vanderrest@students.uu.nl, Utrecht University, The Netherlands.

```
type instance Interpret One               = ()
type instance Interpret Empty             = E
type instance Interpret (Var _ :: IDesc a i) = a
type instance Interpret (dl :∗: dr)       = (Interpret dl, Interpret dr)
type instance Interpret (dl :+: dr)       = Either (Interpret dl) (Interpret dr)
```

Dagand originally defines this universe in a dependently typed setting. Since we choose to use Haskell for our implementation, we need to adapt the original definition slightly, choosing a more restrictive form of the Σ constructor. We do this based on the observation that in most practical use cases of Σ, only the indices of recursive positions depend on the choice for the first element. Since the interpretation of *Var* is independent of the recursive index stored, so is the interpretation of the entire description. Consequently, we can describe this class of functions using a single description with a function $s \to i$ as its index type:

```
Σ :: Proxy s → IDesc a (s → i) → IDesc a i
```

This allows us to interpret a Σ as a regular pair, instead of a dependent pair.

```
type instance Interpret (Σ s d) = (UnProxy s, Interpret d)
```

Here, *UnProxy* maps a value of type *Proxy a* to the type it carries.

DERIVING GENERATORS

Assuming some generator type *G a*, our goal is to define a function of type *IDesc a i* $\to$ *G a* that returns a generator derived from its input description. However, we need a way to describe (at the type level) the dependency between the input description and the type of values generated. It is not possible to encode this relation directly in Haskell's type system, but we can simulate it using *singleton types* [6]. Singleton types bridge the gap between values at the term level and their promoted counterparts, by defining an indexed type that is inhabited by exactly one value for every value of the index type. We define a typeclass *Singleton a* with an associated type *Sing* :: $a \to *$, to capture this notion. We make *IDesc a i* an instance of *Singleton* by defining the type *SingIDesc* (*d* :: *IDesc a i*):

```
data SingIDesc (d :: IDesc a i) where
  SOne   ::                                  SingIDesc One
  SEmpty ::                                  SingIDesc Empty
  SVar   :: i                              → SingIDesc (Var i′)
  (:∗:$) :: SingIDesc l → SingIDesc r → SingIDesc (l :∗: r)
  (:+:$) :: SingIDesc l → SingIDesc r → SingIDesc (l :+: r)
  SSigma :: Proxy s → SingIDesc d → SingIDesc (Σ (Proxy :: Proxy s) d)
```

This enables us to write a function *deriveGen* :: *Sing d* $\to$ *G* (*Interpret d*), which returns a generator whose output type depends on the provided description. We sketch the definition of *deriveGen*, under the assumption that *G* is an instance of both *Monad* and *Alternative*.

```
deriveGen SOne       = pure ()
deriveGen SEmpty     = empty
deriveGen (l :∗: $r) = (,) <$> deriveGen l <∗> deriveGen r
deriveGen (l :+: $r) = Left <$> deriveGen l <|> Right <$> derivegen r
```

In the case of the Σ constructor, we need a way to generate values of the first element type. Since we do not have a general procedure to derive genrators for arbitrary types of kind $*$, we add this generator as a precondition to *deriveGen*. This can be achieved in various ways (e.g. by using typeclasses), so for now we simply assume that a generator $genS :: G\ s$, supplied by the programmer, is in scope. If we now define a type family $Expand$ that transforms descriptions $IDesc\ a\ (s \to i)$ into functions $s \to IDesc\ a\ i$:

$$\textbf{type family}\ Expand\ (d :: IDesc\ a\ (s \to i))\ (x :: s) :: IDesc\ a\ i$$

and a function $expand$ that describes this operation at the term level

$$expand :: forall\ (d :: IDesc\ a\ (s \to i)) \circ Singleton\ s \Rightarrow Sing\ d \to Sing\ s \to SingIDesc\ (Expand\ d\ s)$$

we can define a generator for the Σ constructor by utilizing the monadic structure of G:

$$deriveGen\ (SSigma\ s\ P) = genS \gg\!= \lambda x \to deriveGen\ (expand\ f\ x) \gg\!= \lambda y \to (x, y)$$

completing our definition of $deriveGen$.

GENERATING LAMBDA TERMS

We use the datatype $Term = TVar\ Nat \mid TAbs\ Term \mid TApp\ Term\ Term$ to represent raw terms, and a subset of the representation used in Philip Wadler and Wen Kokke's PLFA [10] to describe well-formedness, restricting ourselves to the constructors for variables, abstraction and application. Let Γ range over contexts and τ over types. A valid judgement of the form $\Gamma \vdash \tau$ can be constructed using one of the following constructors:

$$\texttt{[Var]}\frac{\Gamma \ni \tau}{\Gamma \vdash \tau} \quad \texttt{[Abs]}\frac{\Gamma, \sigma \vdash \tau}{\Gamma \vdash \sigma \to \tau} \quad \texttt{[App]}\frac{\Gamma \vdash \sigma \to \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau}$$

We use a type family $SLTCDesc\ (i :: (Ctx, Type))$ to describe these judgements as a code in the universe of indexed descriptions. We do this by induction over the goal type, return a coproduct of the descriptions corresponding to the constructors that could have been used to arrive at a judgement with that type.

$$\begin{aligned}
&\textbf{type}\ VarDesc = \Sigma\ (P\ CtxPos)\ One \\
&\textbf{type}\ AppDesc = \Sigma\ (P\ Type)\ (Var\ I : * : Var\ I)
\end{aligned}$$

$$\begin{aligned}
&\textbf{type family}\ SLTCDesc\ (i :: (Ctx, Type)) :: IDesc\ Term\ (Ctx, Type) \\
&\textbf{type instance}\ SLTCDesc\ (ctx, T) \qquad\qquad = VarDesc : + : AppDesc \\
&\textbf{type instance}\ SLTCDesc\ (ctx, (t1 : - > t2)) = VarDesc : + : Var\ ((t1{:}ctx), t2) : + : AppDesc
\end{aligned}$$

Given a context ctx and a type ty, $Interpret\ (SLTCDesc\ (ctx, ty))$ is isomorphic to the typing judgement `ctx` ⊢ `ty`, its inhabitants corresponding to raw terms that have type `ty` under context `ctx`. Next, we define an appropriate singleton value $sltcDesc :: Sing\ i \to Sing\ (SLTCDesc\ i)$ to connect the term and type level, along with a conversion function $toTerm :: Sing\ i \to Interpret\ (SLTCDesc\ i) \to Term$ that transforms judgements into terms. Both definitions follow naturally once the type family that describes the inductive relation is in place. By passing the singleton description $sltcDesc$ to $deriveGen$, and applying $toTerm$ to the generated values, we have obtained a generator that produces well-typed lambda terms.

CONCLUSION & FUTURE WORK

We have shown that we can use the techniques described in this abstract to generate well-typed lambda terms, while being reasonably confident that the results are correct. Although a lot is left to be desired in terms of usability and efficiency, we could theoretically synthesize terms of any programming language whose well-formedness can be described using an indexed datatype. Still, there are many possible avenues for further exploration, such as more efficient generation using memoization techniques as used in FEAT [5], property based testing for GADT's, and generation of data whose well-formedness is described by a *mutually recursive* datatypes [8, 11].

REFERENCES

[1] Generation for indexed datatypes. https://github.com/casvdrest/generating-indexed.

[2] Claessen, K., Duregård, J., and Pałka, M. H. Generating constrained random data with uniform distribution. *Journal of functional programming 25* (2015).

[3] Dagand, P.-E. *A Cosmology of Datatypes.* PhD thesis, Citeseer, 2013.

[4] Dénès, M., Hritcu, C., Lampropoulos, L., Paraskevopoulou, Z., and Pierce, B. C. Quickchick: Property-based testing for coq. In *The Coq Workshop* (2014).

[5] Duregård, J., Jansson, P., and Wang, M. Feat: functional enumeration of algebraic types. *ACM SIGPLAN Notices 47*, 12 (2013), 61–72.

[6] Eisenberg, R. A., and Weirich, S. Dependently typed programming with singletons. *ACM SIGPLAN Notices 47*, 12 (2013), 117–130.

[7] Hinze, R., et al. Fun with phantom types. *The fun of programming* (2003), 245–262.

[8] Miraldo, V. C., and Serrano, A. Sums of products for mutually recursive datatypes: the appropriationist's view on generic programming. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development* (2018), ACM, pp. 65–77.

[9] Pałka, M. H., Claessen, K., Russo, A., and Hughes, J. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test* (2011), ACM, pp. 91–97.

[10] Wadler, P., and Kokke, W. Programming language foundations in agda, debruijn: Inherently typed de bruijn representation. https://plfa.github.io/DeBruijn/. Accessed: 2019-06-13.

[11] Yakushev, A. R., Holdermans, S., Löh, A., and Jeuring, J. Generic programming with fixed points for mutually recursive datatypes. In *ACM Sigplan Notices* (2009), vol. 44, ACM, pp. 233–244.