Subtype Polymorphism à la carte via Machine Learning on Dependent Types

Jerry Swan
University of York, UK
jerry.swan@york.ac.uk

Colin G. Johnson
University of Kent, UK
c.g.johnson@kent.ac.uk

Edwin C. Brady
University of St Andrews, UK
ecb10@st-andrews.ac.uk

1 PROBLEM STATEMENT

The essential rationale for subtype polymorphism is adherence to the 'Open/Closed Principle' [12]: the ability to write framework code in terms of superclasses and subsequently invoke it with any subclass that exhibits 'proper subtyping' via the *Liskov Substitution Principle* (LSP) [11]. Formally, the LSP states that if $\phi(t : T)$ is a provable property of objects t of type T, then $\phi(s)$ should be true for objects s of subtype S of T. In practice, such properties have typically been those expressible via 'Design by Contract' [12], specifically preconditions, postconditions and invariants. Such abstraction via subtype polymorphism is intended to insulate against requirements change. However, when new requirements *do* necessitate a change of contract, the maintenance consequences can be severe. In the (typical) absence of explicit language or tool support, enforcement of proper subtyping is laborious and error-prone: contractual changes typically require manual inspection/repair of the class hierarchy to determine/address violations of the LSP.

We therefore claim that the traditional practice of creating top down, *a priori* problem domain abstractions via the LSP is 'upside down' and that there is a useful role for Machine Learning to play in *inducing* these abstractions. More specifically:

(1) Rather than stipulating subtype relationships upfront, they can be reverse-engineered from a codebase.
(2) Such 'just in time reification' can usefully be incorporated into Machine Learning approaches to program synthesis (e.g. [8, 9]).
(3) Induced subtype relationships offer the potential for both increased comprehensibility and asymptotic runtime efficiency.

Below, we outline one specific realisation of the above, in which contracts are specified via dependent types.

2 SUBTYPES VIA DEPENDENT TYPES

For 'Design by Contract' purposes, classes consist of:

- Zero or more attributes.
- An invariant, i.e. a relation on the Cartesian product of the attributes.
- A collection of methods, each having associated operational semantics, as specified by pre- and post- conditions.

```
data Vec2 = MkVec2 Double Double

move: (x, y: Double) → Vec2 → Vec2
move x y (MkVec2 xpos ypos) = MkVec2 (xpos+x) (ypos+y)

data Circle: Vec2 → Type where
  MkCircle: (radius: Double) → Circle position

data Ellipse: Vec2 → Type where
  MkEllipse: (radiusX, radiusY : Double) →
    Ellipse position

moveCircle: (x, y: Double) → Circle pos →
  Circle (move x y pos)
moveCircle x y (MkCircle rad) = MkCircle rad

moveEllipse: (x, y: Double) → Ellipse pos →
  Ellipse (move x y pos)
moveEllipse x y (MkEllipse rx ry) = MkEllipse rx ry
```

Listing 1: Type-level contracts for Circle and Ellipse in Idris

Interpreting the Liskov Substitution Principle via Design by Contract, we can say that some class S is a proper subtype of some superclass T iff:

- S maintains the invariant of T.
- For all overridden methods of T, S "requires no more and ensures no less" than the superclass method it overrides. This is formally expressed as subclass methods having the option of (respectively) weakening/strengthening their pre- and post-conditions.

Dependently-typed languages (such as Agda and Idris) allow families of types to be indexed by *values*. It is therefore possible to express contracts *at the type level* via dependent types (see e.g. Brady [2], Chapter 9). The Idris code of Listing 1 gives a contract for functions *moveEllipse* and *moveCircle* that respectively perform translation of *Circle* and *Ellipse* datatypes. *Vec2* is a type representing 2D vectors, and the declaration $data\ Circle : Vec2 \rightarrow Type$ expresses the fact that *Circle* is dependent on *Vec2*, i.e. there is a distinct *Circle* type for each possible *value* of type *Vec2*. The type signature of the *moveCircle* function thus expresses the postcondition that translating a circle yields a new circle of type indexed by an appropriately translated *Vec2*. Similarly for Ellipse. Listing 2 describes a *Movable* superclass that abstractly defines the contract for translation, with concrete subtypes of *Movable* then defined for *Circle* and *Ellipse*.

3 SUPERTYPE INDUCTION VIA MACHINE LEARNING

We propose that such induction of supertypes could usefully be carried out by a Machine Learning (ML) system. Related work in

ML4PL 2018, July 2018, Amsterdam, Netherlands
© 2018 Copyright held by the owner/author(s).
ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

```
interface Movable (s: Vec2 → Type) where
  move: (x, y: Double) → s pos → s (move x y pos)
  position: s pos → Vec2
  position {pos} s = pos

Movable Circle where
  move = moveCircle

Movable Ellipse where
  move = moveEllipse
```

Listing 2: Induced Movable supertype for Circle and Ellipse

this general area has mined a dependently-typed codebase in search of re-usable proof tactics [10]. For supertype induction, the role of ML is to take datatype definitions such as those in Listing 1 and automatically construct supertypes such as the one in Listing 2.

We envision this as being a component of an integrated compilation/development system; the incorporation of ML (into a role more traditionally played by exact/proof search) places this within the research agenda of *Search Based Software Engineering* [6] which aims to integrate heuristic approaches into the software development workflow. For example, ML could be a background process, where code is constantly scanned for possible supertype abstractions. Alternatively, the developer might assert that certain datatypes belong to a putative supertype, and the system searches for an abstraction.

In terms of the motivating statement about the difficulty of manual maintenance of subtype hierarchies, a key role for the search process is to help maintain consistency as the codebase evolves. When expressed via dependent types, contract failures are recognised at compile time. A ML system of the kind proposed here then has the potential to facilitate automated *repair* of the hierarchy.

To frame a problem in terms of ML, we need to define a search/hypothesis space, and an objective/loss function which specifies the quality of a proposed solution from the hypothesis space. A key role of the objective function is to measure how well a proposed supertype captures the properties of the set of subtypes w.r.t. the LSP, i.e. whether subtypes respect the preconditions, postconditions and invariants of the proposed supertype. The hypothesis space here is that of datatype definitions, which is a combinatorially large function of the size of the codebase (e.g. via consideration of all permutations of equivalent function arguments etc).

We suggest two specific use cases for which a practical search could be done: The first is to explore the space of possible combinations of existing functions (modulo renaming, as appropriate). Particularly for the background search, the objective function would need to penalise ‘useless’ supertype abstractions: one approach would be to require proposed abstractions to have nontrivial preconditions, postconditions and invariants.

```
interface (VerifiedSemigroup a, Monoid a) =>
      VerifiedMonoid a where
  total monoidNeutralIsNeutralL: (l: a) → l <+> Algebra.
        neutral = l
  total monoidNeutralIsNeutralR: (r: a) → Algebra.
        neutral <+> r = r
```

Listing 3: Monoid supertype with contract

The second is to use ML to explore a codebase for possible correspondences with well-known pre-existing abstract types (orderings, monoids, rings, monads etc). and consequently making efficient algorithms (and parallelisations, etc) available. For example, if the abstraction process recognises that some datatype is an instance of a *Monoid* (i.e. an associative binary operator with identity, Listing 3), then exponentiation (i.e. iterated application of the binary operator) can be performed in logarithmic (as opposed to linear) time. By re-factoring *ad hoc* datatypes and associated functions into well-known abstractions there is also the attendant potential for greater human readability.

The choice of specific machine learning methods to be used is open-ended. An important challenge is finding an objective function that guides the search towards interesting supertypes, either by using an interestingness measure [4] or by some form of interactive learning [13] where the user gives feedback on suggestions, and the system builds a model based on that feedback. Furthermore, there may be a link between the hierarchical representation found in layered learning methods such as deep learning [5] and the type hierarchy.

While the induction of supertypes can undoubtedly be framed as a Machine Learning problem, it will necessarily require heuristics that are tuned according to specific use cases. The key research challenge is therefore to devise an objective function for Machine Learning which incorporates human factors concerns for purposes of productive interaction with the developer.

REFERENCES

[1] Edwin C. Brady. 2011. Idris: Systems Programming Meets Full Dependent Types. In *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification (PLPV '11)*. ACM, New York, NY, USA, 43–54. DOI: http://dx.doi.org/10.1145/1929529.1929536

[2] Edwin C. Brady. 2016. *Type-driven Development with Idris*. Manning Publications Company.

[3] Thibault Gauthier and Cezary Kaliszyk. 2014. Matching Concepts across HOL Libraries. In *Intelligent Computer Mathematics*, Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka, and Josef Urban (Eds.). Springer International Publishing, Cham, 267–281.

[4] Liqiang Geng and Howard J. Hamilton. 2006. Interestingness Measures for Data Mining: A Survey. *ACM Comput. Surv.* 38, 3, Article 9 (Sept. 2006). DOI: http://dx.doi.org/10.1145/1132960.1132963

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[6] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based Software Engineering: Trends, Techniques and Applications. *Comput. Surveys* 45, 1 (2012), 11:1–11:61.

[7] Jónathan Heras, Ekaterina Komendantskaya, Moa Johansson, and Ewen Maclean. 2013. Proof-Pattern Recognition and Lemma Discovery in ACL2. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Ken McMillan, Aart Middeldorp, and Andrei Voronkov (Eds.). Springer, Berlin, Heidelberg, 389–406.

[8] Zoltan A. Kocsis and Jerry Swan. 2014. Asymptotic Genetic Improvement Programming with Type Functors and Catamorphisms. In *Semantic Methods in Genetic Programming (SMGP) at Parallel Problem Solving from Nature (PPSN XIV)*, Colin Johnson, Krzysztof Krawiec, Alberto Moraglio, and Michael O'Neill (Eds.). Ljubljana, Slovenia.

[9] Zoltan A. Kocsis and Jerry Swan. 2017. Genetic Programming + Proof Search = Automatic Improvement. *Journal of Automated Reasoning* (Mar 2017). DOI: http://dx.doi.org/10.1007/s10817-017-9409-5

[10] Ekaterina Komendantskaya and Jónathan Heras. 2017. Proof Mining with Dependent Types. In *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*. 303–318.

[11] Barbara Liskov. 1987. Keynote Address - Data Abstraction and Hierarchy. *SIGPLAN Not.* 23, 5 (Jan. 1987), 17–34. DOI: http://dx.doi.org/10.1145/62139.62141

[12] Bertrand Meyer. 1988. *Object-Oriented Software Construction* (1st ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[13] Hideyuki Takagi. 2001. Interactive evolutionary computation: fusion of the capabilities of EC optimization and human evaluation. *Proc. IEEE* 89, 9 (2001), 1275–1296.