Mining opportunities for unique inhabitants in dependent programs

Gabriel Scherer, PhD student, Gallium (INRIA Paris-Rocquencourt)

1 Research Background

My ongoing work focuses on types that have a unique inhabitant – modulo program equivalence. If we were able to detect when such types appear in a program, we could infer the corresponding code, releaving the programmer from the obligation to write the less-interesting parts of the program.

The long-term goal is to be able to detect types with unique inhabitants, and provide programmers of typed languages with a new feature, a joker that can be used as a term (of any type), and is given a meaning after the program has been type-checked. If the expected type at this point has a unique inhabitant, elaborate into it, otherwise fail by warning the programmer of the ambiguity. Note the contrast with proof-assistant automation that is only concerned with *inhabitation*, as the dynamic identity of proof witnesses is neglected.

Our speculation is that this feature would be particularly useful for the "boring" parts of a program, the glue that is around the places where the real semantics is expressed; when you feel bored at the idea of writing some part of your code, it's probably because the compiler could write it for you.

We think that the availability of this feature, and a correspondingly adapted type-system design , would promote different programming styles, alternating code and type inference, with more types with unique inhabitant than in existing code. Before investing considerable efforts in this direction, however, it is important to look at the existing codebases to check that this wouldn't remain purely anecdotical.

2 Proposed talk

In this talk I propose to discuss the occurrence (or lack thereof) of types with unique inhabitants in existing dependently-typed programs. I spent some time looking at the research literature for examples of programs written in dependently typed languages (mostly Coq and Agda, but not only), looking at whether program fragments could have been inferred by our hypothetical joker.

The conclusions are that it very much depends on the programming style. In broad strokes, we could say there is a split between programs that are "implemented, then proved correct", and those that are "implemented correctly" in a single step. The former will not necessarily have much stronger types than in a ML or Haskell implementation, and draw their safety from an adjoined lemma. The latter will use refined types directly in the implementation, providing strong invariants, and more opportunities of types with a unique inhabitant. There is an ongoing debate and experimentation about which of these styles should be preferred when writing programs – and more generally software engineering principles for dependently typed programming. We think program-inference tools could add an interesting perspective.

Finally, we were surprised to find allies among the seemingly "less dependent" layered languages of the LF tradition, in particular VeriML. In those languages, the most powerful typing features are restricted to a second class of "logic objects" which do not include the general computation mechanisms of the host language. A consequence of this layering is that the mathematical assertions expressible in the language do *not* talk about the computations defined in the same program; the first style of implementing then proving correct is not available.

For this reason, VeriML programs are troves of opportunities for code inference directed by types with unique inhabitants — restricted by the type of type witnesses built along computations. Interestingly, the VeriML designers have not considered code inference from the logical types, but instead emphasized the possiblity of stronger type inference in the future; relying on the witness construction code (that we would allow to omit) to remove type annotations.

(This page contains non-essential appendices that can be skipped for review.)

A Sources

We would like to demonstrate concrete examples taken from existing dependently typed programs, as described in the following papers. There is a definite tropism towards programs that manipulate programs (type checkers, syntax representation, proof automation...), which we think comes from the available literature rather than a selection bias.

References

[BHKM09] Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly Typed Term Representation in Coq. 2009.

[Cha08] James Chapman. Type Theory should eat itself. 2008.

[Jef12] Alan Jeffrey. LTL types FRP. 2012.

[Sta13] Antonios Stampoulis. VeriML: A dependently-typed, user-extensible and language-centric approach to proof assistants. 2013.

B Related work, and a quote

There is existing work in the setting of interactive tooling (IDEs, etc.), doing type-directed name completion for example. In these works, the idea is more to provide lots of different suggestions, using the right scoring heuristics so that the user's final choice is, on average, in the first few results. This is vastly different from our principled (and correspondingly less often applicable) inference, based on absolute certitude of the correction of the choice, which also makes it suitable in non-interactive workflows, or without user validation: we are not *required* to show to the programmer the skeleton elaborated by our joker before the program execution.

We are aware that using types for code inference is an existing idea that is maybe part of the folklore of the dependent type community, as eloquently described by Conor McBride in 2011 – quote attached below. However, to our knowledge, the work on unique inhabitants is sparse, and has not yet found practical applications for program writing, in dependently typed systems or otherwise.

Conor McBride, "Why do programming languages use type system?"
`http://www.quora.com/Why-do-programming-languages-use-type-systems`

> Whilst I don't want to gainsay the importance of types as a source of corrective raspberry-blowing, I would like to offer the prospect that types might have an active role to play, structuring the process of program inference. Overloading allows you to get rid of boring lumps of code if it can be figured out from types. Datatype-generic programming uses representations of the structure of types to calculate specific instances of algorithm-schemes. Dependent type systems often allow run-time-relevant values to be inferred silently from type information.
>
> This position constitutes a change of viewpoint in the purpose of types. If programs worked just the same with the types rubbed out, then types would represent a form of piety often inadequate with respect to testing. It's when types contribute information to algorithm selection, design statements which program definitions need merely refine, that they constitute a significant win.
>
> To be fair, even in last century's typed languages, types had a beneficial organisational effect on programmers. This century, it's just possible types will have a comparable effect on programs. Types are concepts and now mechanisms supporting program-discovery as well as error-discovery.