Adding Soft Types to Isabelle

Alexander Krauss

2010

This note describes how I would like to approach the "soft type challenge" in Isabelle. It collects ideas that emerged in discussions with many people including Jeremy Avigad, John Matthews, Tobias Nipkow, Larry Paulson, Andreas Schropp, and Makarius Wenzel. Some of these ideas were previously summarized by notes by Larry Paulson (in 2007) and Jeremy Avigad (in May 2010). Here, I am trying to be more specific and concrete where possible, and to outline a concrete research program.

Slogan: This work is not about inventing a logic or type theory. This is about making sets more convenient to use than types are today, by building powerful tools.

1 Motivation

Almost all major proof assistants are based on some form of *type theory*, where the concepts of *type* and *type checking* are fundamental primitives that are fixed once and for all. While types have many advantages, using them as a foundation deviates from the de-facto standard in mathematics, where a set-theoretic foundation is taken for granted. Moreover, types can become a serious limitation when the studied concepts are not easily expressible within their limits. There are many examples of theories that have to be tuned significantly to make them compatible with the type discipline.

- Proof assistants based on higher-order logic cannot express the type of all $n \times n$-matrices naturally, since types may not depend on values.
- While dependent type theories (e.g., CIC) can express strong specifications within the type system, their handling of equality is problematic: Definitional equality, which is typically used for type checking, is quite weak. For example, the types $T(1 + n)$ and $T(n + 1)$ are regarded as different in an intensional type theory.

  In particular, the definition of some object is treated in a fundamentally different way than other properties of the object which are derived from the definitions. This makes it harder to build abstractions, where objects are characterized in terms of their properties and not by their construction. Here, the extensional type theory of the NuPRL system is a notable exception.

- Many proof assistants cannot express subtyping naturally, even though it is very natural to assume that, e.g., the different sets of numbers are subtypes of one another: $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C}$. Moreover, subtyping is useful to express invariants of data structures.
- While the Isabelle system has a native implementation of type classes as part of the foundation, it lacks useful extensions such as multi-parameter type classes and constructor classes.

Most of these concrete deficiencies of existing systems can be addressed by modifications and extensions to the type system. However, in practice this rarely happens, because the type system is rigidly connected with the logical foundations of the theorem prover. Changing it raises meta-theoretical issues about the soundness of the modified system, and also requires significant modifications to the implementation of the theorem prover, which typically renders existing developments incompatible. Moreover, there is no single system that handles all these issues well.

As a result of this dilemma, users of theorem provers accumulate elaborate tricks to partly circumvent certain limitations. For example, Harrison's formalization of n-ary cartesian products [2] encodes the number n as a type (which seriously complicates induction proofs over n), and Huffman et al. encode constructor classes as values in HOLCF [3] (the approach was never actively used, due to its complexity). Such tricks can be useful, but they are more often in the way.

2 Our vision: soft types

This project will take a radically different approach: Rather than hard-wiring types into the calculus, we aim to create a proof assistant with "soft types", as a flexible and extensible infrastructure on top of untyped set theory.

In set theory, the equivalent of a typing judgement $a : A$ can be expressed as set (or class) membership assertion $a \in A$. Thus, types become first-class citizens, and a "type checker" is nothing more than a procedure that proves statements about sets automatically. Depending on the application, this can be a straightforward implementation of simply-typed lambda calculus, or a more sophisticated tool, which allows for dependent types, and incorporates other automated reasoning procedures.

This approach has the potential to result in a proof assistant which is more powerful and more flexible than all systems currently in existence.

In addition to the advantages of typed systems, we gain

Power More expressive forms of typing can be added by users or libraries.
Types are always explained in terms of the underlying set theory, which

gives these extensions a clear semantics. Since type checking is not different from other reasoning, existing automated proof tools can be used.

Flexibility For different applications, or in different places within a single application, different forms of typing may be used. This is important, since no single procedure can suit all applications.

Safety Proofs produced by the type checker are treated in the same way as other proofs: the system's minimal trusted kernel checks them for correctness. Thus, errors in the type checker cannot accidentally make the whole system unsound.

Simpler Foundation The logical foundation of the whole system becomes much simpler, and closer to standard mathematics.

3 Basic Approach

Since introducing soft types touches many areas of the system, it is hard to find a place to start. It is important not to do too many things at once. The following route may be a suitable way to get started.

1. Represent soft types in Pure as predicates (of type $\alpha \Rightarrow \mathit{prop}$, or maybe $\alpha \Rightarrow \mathit{bool}/o$). Π-types can be expressed in Pure already, whereas Σ-types can only be added in the object logics, since they depend on a notion of pair.

2. Implement a simple type checker as a proof procedure similar to the `typecheck` tactic in ZF, but in a way that it can solve systems of constraints. Like `typecheck`, it will use information declared in the context.

   This type checker is only a basis for experimenting with the framework, and we will probably refine it continuously as we go on, and re-do it several times. Being able to do this flexibly is the main advantage of soft types.

3. Using the existing `check/uncheck` infrastructure by Makarius, implement implicit arguments, which are inserted with the help of the type checking procedure. While the general idea of implicit arguments seems simple, there may be some tricky bits in the implementation, and documentation on this seems to be weak. We might want to interview some people that know about the implementation in other systems (Coq/Agda/Matita/?).

4. Build variants of the `rule`, `erule` tactics that solve typing assumptions automatically. This is to support single-step apply-style proofs without typing conditions getting in the way.

5. Instrument a few automated tools to use type information from the context. We might start with `simp` and `auto`. Other tools like `blast` should probably be ignored for the moment; at a later stage we may need a general way of dealing with type-unaware tools, but this requires experimentation.

At this point, we have gained basic soft type checking and convenient syntax through implicit arguments. The implicit syntax is important: it hides type arguments from polymorphic functions and can later help to resolve ambiguous notation and hide dictionary arguments when a type class mechanism is added.

Ideally, most of this can be done generically in Pure and instantiated for object logics. This lets us experiment with the tools in HOL, where we can play with existing formalizations, and then move on to ZF, which is our actual target.

4 Case Studies

We should play with applications early in the process. A few things immediately come to mind, in increasing difficulty.

- **Vectors** of length n (simple dependent type). We should be able to state and prove associativity of append without running into the typing issues commonly encountered in Coq. This probably requires including the simplifier into the type checking phase.
- As a slight generalization, we can look at $n \times m$**-matrices**, which have been challenging in HOL: While the set of $n \times m$ matrices forms a ring, the set of *all* matrices does not, since there is no finite matrix that is a multiplicative unit for all dimensions. With soft types the situation is better, since we can specifically consider $n \times m$ matrices.
- Also related to vectors of fixed length, we should see if we can get a faithful model of **Cryptol's sized types**. This could considerably improve the shallow embedding of Cryptol in Isabelle.
- **Constructor Polymorphism**. This requires ZF. Monads can be modelled generically by specifying a set operator M in a locale. This is already possible today, but it is not practical, since there is just too much typing clutter in the way. Soft types should be able to hide most of this.
- **Subtypes**. This can be predicate subtypes in the PVS sense, but we should experiment with other systems: In Mizar, types can be modified by "attributes" (e.g., `nonzero`, `positive`, `nonempty`). It seems that by declaring such subsets explicitly and by giving appropriate rules, one can avoid generating arbitrary type checking conditions, which would arise from the general $\{x \in A \mid P(x)\}$. Here we must experiment and learn from Mizar.

5 Challenges

5.1 ZFisms

A few specific issues arise when using ZF.

Sets vs. classes It is natural to permit that types can be classes, which arises from the use of predicates in Pure. It also allows us to write things like the type of all groups, as a dependent tuple containing the carrier set and the operations.

In some situations, types may need to be "internalized" into sets. For example, the type of all groups can be restricted to the set of all groups whose carrier is a subset of the natural numbers. We must find out when this is necessary, and provide tools that make the conversion smooth. However, this is not entirely new: even in HOL, one regularly applies *atomize* and *rulify* conversions to expressions, which convert meta logic to object logic and back. The class/set transition could be similar.

Object vs. meta functions This is essentially the same issue as above, just for functions instead of relations. As noted by Larry, working with meta functions is more convenient, so we should try to stick with them whenever possible. Note that meta functions can still be typed in the Pure soft type framework. If for a function $f :: i \Rightarrow i$ we use the syntax $f :: [A] \Rightarrow [B]$ to express that $\forall x \in A. f(x) \in B$ then the application operator ‘ and the ZF abstraction operator *Lambda* can be soft-typed as

$$` :: [A \to B] \Rightarrow [A] \Rightarrow [B] \ .$$

$$\mathit{Lambda} :: [A] \Rightarrow ([A] \Rightarrow [B]) \Rightarrow [A \to B] \ .$$

This also works in the dependently typed versions.

5.2 Type Inference

If we want to support some sort of dependent types or even just a combination of polymorphism and subtyping, we must end up with general type inference being undecidable.

But there is hope that a reasonably well-behaved partial type inference can be built, which only rarely requires explicit annotations. In dependent type theories this is common practice and successful, although the details are tricky, and not very well-documented. A good source of information is a recent draft by Pientka, who had to implement type reconstruction for LF, and documented some tricky parts in good detail [7]. Norell's thesis [5] describes the inference mechanism used in Agda. It is interesting and potentially helpful that a similar mechanism already exists in Isabelle: The reconstruction process which restores the information elided from proof terms is nothing but a type inference mechanism for the language of proof terms.

On the programming language side, the situation is similar: full type reconstruction for polymorphism plus subtyping is undecidable, but using local type inference [8, 6], it seems that practical inferences can be built. The Scala language uses them successfully.

Interestingly, in both worlds, some form "bidirectional" type checking algorithm is used, even though the type systems are quite different.

5.3 Isar Integration

The basic setup can be implemented without making any fundamental changes to the system's architecture. For initial experiments, this is acceptable but it has a few limitations. Most importantly, it requires that the type of all variables is written explicitly in the Isar text:

```
fix $x$
assume [type]:  $x \in A$ [1]
...
```

Similarly, the typing assumptions will be visible in top-level theorem statements. For more concise specifications and proofs, we would like to apply type inference here.

A consequence for the Isar architecture is that *reading a term may extend the context with additional assumptions* like $x \in A$ above. In principle, the existing `declare_term` operations could be used to implement this, but at the moment they only add non-logical constraints to the context that affect subsequent type inference. Reinterpreting these operations to do anything logically significant will require a careful reexamination of the places where it is used. Similar "refactorings" happen regularly in the normal Isabelle development process, but they require time and significant effort.

We therefore postpone these issues until we have a clearer understanding of the other components of the infrastructure, and until we have done some of the case studies.

5.4 Structures and Type Classes

While it is not new that a combination of implicit arguments and clever type inference can implement type classes (Coq does it this way [9], and Scala does too [1]), we still do not know how to combine the parts into a coherent whole. In Coq, there seem to be multiple ways of using type classes to model algebraic structures. While both techniques use some form of dependent tuple/record, the question is which components are modelled as record components and which are parameters. Spitters and van der Weegen [10] explore these design considerations in Coq.

We will have to address similar questions in the context of Isabelle and soft types. The approach using unification hints mentioned in Avigad's notes seems to be one way of doing it, but there may be others.

5.5 Users and Libraries

We cannot expect to build a system that can replace Isabelle/HOL within the next years. Thus, the soft type project will remain a minority interest for some time. As pointed out by Avigad, this can be an advantage, since we

[1]The more concise `fix` $x \in A$ suggests itself, of course.

can concentrate on the fundamental issues without worrying about breaking anybody's large developments. However, at some point, significant porting of theories will be required.

However, given the excitement that the soft type discussion often generates, I have no doubt that once we get the fundamentals of the system right, it will be easy to find people to help with the porting. Recent ports of theories from HOL Light to Isabelle show that porting theories is hard but feasible, and the results are often better than the original.

References

[1] B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *OOPSLA/SPLASH 2010*, 2010. To appear.

[2] J. Harrison. A HOL theory of euclidean spaces. In Hurd and Melham [4], pages 114–129.

[3] B. Huffman, J. Matthews, and P. White. Axiomatic constructor classes in Isabelle/HOLCF. In Hurd and Melham [4], pages 147–162.

[4] J. Hurd and T. F. Melham, editors. *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*. Springer Verlag, 2005.

[5] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.

[6] M. Odersky, M. Zenger, and C. Zenger. Colored local type inference. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 41–53, 2001.

[7] B. Pientka. An insider's look at lf type reconstruction: Everything you (n)ever wanted to know. submitted, Aug 2010.

[8] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000.

[9] M. Sozeau and N. Oury. First-class type classes. In O. Ait Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer Verlag, 2008.

[10] B. Spitters and E. van der Weegen. Developing the algebraic hierarchy with type classes in Coq. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving (ITP 2010)*, volume 6172 of *Lecture Notes in Computer Science*, pages 490–493. Springer Verlag, 2010.