Toward SMT-based Refinement Types in Agda

GAN SHEN, University of California, Santa Cruz, USA
LINDSEY KUPER, University of California, Santa Cruz, USA

Dependent types offer great versatility and power, but developing proofs with them can be tedious and requires considerable human guidance. We propose to integrate Satisfiability Modulo Theories (SMT)-based refinement types into the dependently-typed language Agda in an effort to ease some of the burden of programming with dependent types and combine the strengths of the two approaches to mechanized theorem proving.

1 INTRODUCTION

Agda [Norell 2008] is a dependently-typed programming language where types can contain (depend on) terms. With the expressivity of dependent types, we can encode rich, precise properties of programs using types and prove that those properties hold at compile time through type checking. A standard example of a dependent type is the type of lists of a given length: `Vec A n`. Here `A` is the type of list elements and `n` is the length of the list. Then we can write a `lookup` function of type `Vec A n → Fin n → A` where `Fin n` is the type of natural numbers less than `n`, and rest assured that there will never be an out-of-bounds list index error at run time.

However, the power of *full* dependent types comes at a price — writing programs amounts to writing proofs, and proofs can be difficult and tedious to write. For instance, suppose we want to use the aforementioned `Fin n` in a context where `Fin (n + 1)` is required. Even though something of type `Fin n` should be usable in that context, the Agda type checker complains, forcing us to write a proof to show that a term of type `Fin n` can also be typed `Fin (n + 1)`. This problem could be avoided if we had Satisfiability Modulo Theories (SMT)-based refinement types [Rondon et al. 2008; Swamy et al. 2016; Vazou et al. 2014, 2017], which have a subtyping rule that uses an off-the-shelf SMT solver to automate the proof for us. We can think of refinement types as a restricted form of dependent types in which types can contain terms drawn from a decidable logic. As a result, proofs can be done automatically by SMT solvers, making certain programs easier to write.

In this paper, we propose to integrate SMT-based refinement types into the dependently-typed language Agda in an effort to ease the burden of programming with dependent types. We illustrate some of the challenges of using dependent types, discuss how refinement types can alleviate some of them, and consider implementation approaches.

2 THE BURDEN OF PROOF

To motivate the need for refinement types in Agda, we consider a simple example from the Agda standard library [Danielsson et al. 2021a] (with some tweaks to aid readability). The type `Fin n` denoting natural numbers less than `n` is inductively defined as:[1]

```
data Fin : ℕ → Set where
  fzero : {n : ℕ} → Fin (suc n)
  fsuc  : {n : ℕ} (i : Fin n) → Fin (suc n)
```

[1]In Agda, arguments wrapped in curly brackets are implicit arguments, which means we can omit them when calling the function because the type checker can infer their values.

Authors' addresses: Gan Shen, University of California, Santa Cruz, USA; Lindsey Kuper, University of California, Santa Cruz, USA.

Here `Fin n` is a data type indexed by a natural number[2] `n`. Elements of `Fin n` can be seen as natural numbers in the set $\{m \mid m < n\}$.

We can understand this data type definition as saying:

- For all natural numbers `n`, `fzero` is a member of the set `Fin (suc n)`.
- For all natural numbers `n` and `i` in set `Fin n`, `fsuc i` is a member of set `Fin (suc n)`.

Even though semantically `Fin n` is just a specialized $\mathbb{N}$, because it is a whole separate definition, one cannot use it in a context where a $\mathbb{N}$ is required. This could potentially lead to unnecessary verbose proofs and poorly performing code. For example, consider the predecessor function `pred` for `Fin n`, also from the Agda standard library, which assumes the predecessor of `fzero` is still `fzero`:

```
pred : {n : ℕ} → Fin n → Fin n
pred fzero    = fzero
pred (fsuc i) = inject₁ i
```

Everything looks pretty normal except that we have to call this mysterious $\texttt{inject}_1$ function. To understand what is happening, let's take a close look at how the `fsuc i` case is type checked. Due to the type of `fsuc`, we know that `fsuc i` must have type `Fin (suc n′)` for some `n′`, so we know that `n = suc n′` and our end goal changes to `Fin (suc n′)`.

To get the predecessor of `fsuc i`, we want to simply drop the `fsuc` and return `i`, but `i` has type `Fin n′` as opposed to `Fin (suc n′)`, as required. Naturally, a natural number less than `n′` is also less than `suc n′`, but Agda stubbornly rejects that. Instead, we need to provide a proof saying a `Fin m` is also a `Fin (suc m)`, which is called $\texttt{inject}_1$ in the Agda standard library:

```
inject₁ : {m : ℕ} → Fin m → Fin (suc m)
inject₁ fzero    = fzero
inject₁ (fsuc i) = suc (inject₁ i)
```

$\texttt{inject}_1$ takes a `Fin m` and lifts it to `Fin (suc m)` while keeping the underlying data structure intact. From a programming perspective, $\texttt{inject}_1$ does nothing interesting, but has time complexity $O(n)$. It is acceptable if we never intend to actually execute the code, but would be vexing for those who want to extract a efficient implementation from Agda or who just want to do practical programming in Agda.

3 REFINEMENT TYPES TO THE RESCUE

Having seen how cumbersome it is to use the inductively defined `Fin n` type in Agda, we turn to a different encoding that uses subtype polymorphism and an underlying SMT solver to ease the burden of programming. A natural number less than `n` is really just a subset of natural numbers. With refinement types, we can say:

$$Fin\ n \triangleq \{x : \mathbb{N} \mid x < n\}$$

In general, base refinements of the form $\{x : B \mid P(x)\}$ are the building blocks of refinement types. Here B is a basic type, and P is the refinement predicate constraining the value x. This refinement type denotes a subset of B whose elements satisfy the refinement predicate P. Thus, a basic type B without refinement can be thought of as an abbreviation for $\{x : B \mid \top\}$.

We often find ourselves wanting to use a more specific type in a more relaxed context — as in the above `pred` example, where we want to use `Fin n′` in a context where `Fin (suc n′)` is required

[2] $\mathbb{N}$ is the type of natural numbers in Agda. Following Peano arithmetic, it has two constructors, `zero` and `suc`. Agda also supports decimal notation, so one can write `2` for `suc (suc zero)`.

— so we define a subtyping rule for refinement types:

$$\frac{[\![\Gamma, x : B]\!] \vDash [\![P(x)]\!] \Rightarrow [\![Q(x)]\!]}{\Gamma \vdash \{x : B \mid P(x)\} \preceq \{x : B \mid Q(x)\}} \quad \preceq\text{-Base}$$

The $\preceq$-Base rule says that the base refinement $\{x : B \mid P(x)\}$ is a subtype of $\{x : B \mid Q(x)\}$ under the context Γ if and only if $[\![P(x)]\!]$ implies $[\![Q(x)]\!]$ under the context $[\![\Gamma]\!]$ extended with $[\![x : B]\!]$. Here $[\![\cdot]\!]$ means the translation from Agda terms to SMT terms.

For example, the subtyping relation:

$$n : \mathbb{N} \vdash \{x : \mathbb{N} \mid x < n\} \preceq \{x : \mathbb{N} \mid x < (suc\ n)\}$$

follows from the validity of the implication:

$$n : \mathbb{N},\ x : \mathbb{N} \vDash x < n \Rightarrow x < n + 1$$

which tells us *Fin n* is a subtype of *Fin* (*suc n*). This is what we need to be able to infer a sufficiently specific return type for `pred`.

4 IMPLEMENTING REFINEMENT TYPES IN AGDA

A refinement type $\{x : B \mid P(x)\}$ can be readily emulated in a dependent type system as the Σ (dependent pair) type $\Sigma_{x:B} P(x)$, where B is a type and P is a function from B to a type. For readability, following the Agda standard library [Danielsson et al. 2021b], we define a record type `Refinement` in Agda and give it a convenient syntax resembling set comprehension:

```
record Refinement (B : Set) (P : B → Set) : Set where
  constructor _,_
  field
    value  : B
    .proof : P value

syntax Refinement B (λ x → P) = [ x ∈ B | P ]
```

A value of type `Refinement` is a pair of `value` and `proof`, where `value` is the basic type being refined and `proof`[3] is the refinement. To construct a value of refinement type, we use the `_,_`[4] constructor and provide both value and proof:

```
2≡2  : [ x ∈ ℕ | x ≡ 2 ]
2≡2  = 2 , refl

2≤10 : [ x ∈ ℕ | x ≤ 10 ]
2≤10 = 2 , s≤s (s≤s z≤n)
```

A function that operates on values of refinement type can extract the value and proof from the argument and use them to build the result:

```
pred : {n : ℕ} → [ x ∈ ℕ | x < n ] → [ x ∈ ℕ | x < n ]
pred (zero  , p) = zero , p
pred (suc x , p) = x    , <-trans (n<1+n x) p
```

[3]The dot syntax marks the `proof` component of `Refinement` as *proof-irrelevant*, which prevents us from using the proof in computation; we discuss this further below.

[4]Agda supports defining mixfix operators using underscores _ to mark where arguments are positioned when applied, so we can apply `_,_` to `a` and `b` by writing `a , b`.

The refinement type syntax is convenient, but the real advantage of SMT-based refinement type systems like Liquid Haskell [Vazou et al. 2014] and F* [Swamy et al. 2016] is that they automate away much of the burden of having to explicitly construct the proof. Instead of explicitly writing proofs like `refl`, `s≤s (s≤s z≤n)`, and `<-trans (n<1+n x) p`, we can ask an SMT solver to carry out the proof for us.

Furthermore, if such a proof does not exist — that is, if the SMT formula translated from the Agda refinement predicate turns out not to be valid — the solver can provide a counterexample, corresponding to a model that satisfies the negation of the formula. Such a counterexample could provide more insight into the problem than a plain type mismatch error message.

To bridge between Agda and SMT, we plan to use Schmitty [Kokke 2021], an Agda library that provides bindings for SMT-LIB [Barrett et al. 2021] (the common input language used by all mainstream SMT solvers) compatible solvers and macros for translating Agda terms to SMT-LIB scripts. Combined together, these features can enable automatic proving. In particular, Schmitty provides a macro `solveZ3` that calls an external SMT solver during type checking. `solveZ3` works by:

(1) Getting the typing context Γ and the expected type τ from its call site.
(2) Translating the implication $\Gamma \implies \tau$ to SMT-LIB script and asking if the negation of the formula is unsatisfiable (which is equivalent to the validity of the original formula in classical logic; we will discuss the interplay between this use of the solver and Agda's constructive logic below).
(3) Inserting a dummy proof (a *postulate* in Agda) at the call site if the solver deems the negated formula to be unsatisfiable, and otherwise reporting an error (counterexample).

With the help of the `solveZ3` macro, the above code examples can be simplified as follows:

```
2≡2  : [ x ∈ ℕ | x ≡ 2 ]
2≡2  = 2 , solveZ3

2≤10 : [ x ∈ ℕ | x ≤ 10 ]
2≤10 = 2 , solveZ3

pred : {n : ℕ} → [ x ∈ ℕ | x < n ] → [ x ∈ ℕ | x < n ]
pred (zero  , _) = zero , solveZ3
pred (suc x , _) = x    , solveZ3
```

We can introduce further macros that hide the proof component of values of type `Refinement`. The resulting code feels similar in spirit to programming in a mainstream functional programming language like Haskell or OCaml, yet programmers still have access to the full power of dependent types if need be. Tricky proofs that cannot be automated by the underlying SMT solver can still be carried out in Agda by hand.

There are disadvantages to using an SMT solver to automate proofs in Agda. Two that are especially worthy of mention are:

- The dummy proof generated by `solveZ3` doesn't have computational meaning. We mitigate this by annotating the proof component of `Refinement` as *proof-irrelevant*, which prevents us from using the proof in computation, so whether it has computational meaning doesn't matter. It's still an open problem to extract proofs from a solver and reify them into Agda [Barrett et al. 2015].
- SMT solvers reason in classical logic, so by using them to automate proofs in Agda, we are essentially strengthening Agda's logic from constructive to classical. That said, it is already possible to constructively prove an excluded-middle property (without the use of SMT) for many of the predicates we are working with (e.g., for equality of natural numbers, we can

prove that for all natural numbers x and y, `x ≡ y ⊎ ¬(x ≡ y)`), so it can be argued that in such cases SMT only introduces automation rather than changing the underlying logic.

5 CONCLUSION AND FUTURE WORK

Theorem-proving systems have historically fallen into two camps [Boutin 1997]: interactive theorem proving, exemplified by proof assistants such as Agda, and automated theorem proving, exemplified by SMT solvers and tools that build on them. Each approach has its strengths and weaknesses. SMT-based tools provide "push-button" operation: given a proposition, they return either true or false without human guidance. They are great as far as they go, and can automate many tedious proofs, but they offer little feedback to the user. On the other hand, dependent-types-based tools provide great versatility and power, but developing proofs with them requires considerable human guidance.

The boundary between the two camps is blurred by hybrid automatic/interactive (sometimes called "auto-active" [Leino and Moskal 2010]) verification approaches, with different tools providing varying degrees of granularity of feedback provided to the human user, and different styles of specifying program properties. A version of Agda augmented with SMT-based refinement types would represent one point in this space, as do emerging proof assistants that integrate SMT automation such as Lean [de Moura et al. 2015] and F* [Swamy et al. 2016]. Unfortunately, the research community's knowledge of such hybrid verification tools is fragmented: they are dots on a map, but we have no robust theory to tie them together, leading to the tools being poorly understood and hence under-exploited. As a starting point, we propose that integrating SMT-based refinement types into Agda would result in a hybrid verification tool in which the strengths of SMT-based and dependent-types-based tools can be combined. In the long run, we hope to not explore not only this one point in the design space, but to systematically survey the landscape of such hybrid tools and contribute to a holistic scientific understanding of the space.

REFERENCES

Clark Barrett, Leonardo De Moura, and Pascal Fontaine. 2015. Proofs in satisfiability modulo theories. *All about proofs, Proofs for all* 55, 1 (2015), 23–44.

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2021. The Satisfiability Modulo Theories Library (SMT-LIB). http://smt-lib.org.

Samuel Boutin. 1997. Using Reflection to Build Efficient and Certified Decision Procedures. In *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan, September 23-26, 1997, Proceedings (Lecture Notes in Computer Science)*, Martín Abadi and Takayasu Ito (Eds.), Vol. 1281. Springer, 515–529. https://doi.org/10.1007/BFb0014565

Nils Anders Danielsson, Matthew Daggitt, and Guillaume Allais. 2021a. The Agda standard library: Finite sets. https://agda.github.io/agda-stdlib/Data.Fin.Base.html.

Nils Anders Danielsson, Matthew Daggitt, and Guillaume Allais. 2021b. The Agda standard library: Refinement type. https://agda.github.io/agda-stdlib/Data.Refinement.html.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *International Conference on Automated Deduction*. Springer, 378–388.

Wen Kokke. 2021. Schmitty. https://github.com/wenkokke/schmitty

K Rustan M Leino and Michał Moskal. 2010. Usable auto-active verification. http://fm.csl.sri.com/UV10. In *Usable Verification Workshop*.

Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming (AFP'08)*. Springer-Verlag, Berlin, Heidelberg, 230–266.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin.

2016. Dependent Types and Multi-Monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 256–270. https://doi.org/10.1145/2837614.2837655

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *Proc. ACM Program. Lang.* 2, POPL, Article 53 (Dec. 2017), 31 pages. https://doi.org/10.1145/3158141