Extended Abstract: Developing a Dependently Typed Language with Runtime Proof Search

Mark Lemay
Cheng Zhang
William Blair
Computer Science
Boston University
USA
lemay@bu.edu

Abstract

The Curry-Howard correspondence identifies functions with theorems, providing a promising link between well explored areas of math and software engineering. However dependently typed systems can be cumbersome when used as a programming language. We believe that one solution is to extend dependently typed languages with a proof search interface that can be activated at run-time, delaying constraint solving as late as possible. We describe insights gained from ongoing work implementing a prototype Call-By-Value dependently typed language with these run-time solving capabilities that should help other teams trying to implement something similar.

Keywords Programing Languages, Dependent Types

1 Introduction

There are many attempts to make dependent types more usable by discharging simple lemmas: through tactics (Coq or lean), or proof search procedures that are built into the development environment (Agda). For propositions that are not easily solvable, the only other options are a manual proof or `assert`ing the lemma. `assert`ing propositions is a risky strategy, since there will be no automatic indication when an `assert`ed proposition is unreasonable. Previous authors have tried to solve this issue by building a QuickCheck[Claessen and Hughes 2011] like system in Coq called QuickChick[Dénès et al. 2014] to test assertions. Other authors have explored testing and SAT solving in a previous version of Agda[Haiyan 2003].

The Curry-Howard correspondence naturally identifies lemmas and helper functions. In existing implementations, lemmas and helper functions defined with `assert` are not given run-time behavior. We think there is a missed opportunity to turn these assertions into a run-time search that gives a concrete value of the type asserted.

Conference'17, July 2017, Washington, DC, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00

2 Theory

Type Soundness (in the sense of progress and preservation) generally holds, since `assert` has a type equivalent to $\Pi A : \textbf{\textit{type}}.A$, which is assumable in most dependent type theories. However any definitional behavior of `assert` must be consistent with the standard lemmas for Type Soundness. For example, `assert` must behave functionally, if $\texttt{assert}[\mathbb{N}] \equiv 0$ and $\texttt{assert}[\mathbb{N}] \equiv 1$ in the same system, then $\vdash \texttt{refl}(\texttt{assert}[\mathbb{N}]) : \texttt{Id}(0, 1)$ and reasonable notions of preservation will not hold.

Obviously, this system is not sound in the logical sense (every type is inhabited in the empty context). However we believe a partial correctness property (like that advocated in [Jia et al. 2010; Sjöberg et al. 2012]) holds. For instance, a term of type $\Sigma n : \mathbb{N}.\texttt{isEven}(n)$ may not terminate, but if it does evaluate to a value, the first part of the pair will be even.

Since there are possibly non-terminating terms, statically evaluating `assert`s can cause type checking to be undecidable. This gives our type theory a character like [Jia et al. 2010]. Since type checking is undecidable and there are non-terminating terms, we included convenient syntax for recursive functions like in [Jia et al. 2010].

Since we use a Call-By-Value evaluation strategy `assert`s can be used to block problematic computation. For example, $\texttt{head} = \lambda ls.\textbf{\textit{let}}\, pr = \texttt{assert}[\texttt{len}(ls) > 0]\textbf{\textit{in}}...$. Though type checking is undecidable, we believe that we can always insert assertions such that type checking goes through and evaluation will only complete if the types are compatible, while always maintaining a consistent result of computation.

3 Practice

Solving for inhabitants of types in most dependently typed theories is arbitrarily hard, and so an adequate theory of `assert`s is unlikely to result in a useful run-time system by itself. To improve this, we propose a passive testing system that will precompute assertions and warn if assertions are hard to derive at concrete values (which is an indication that an incorrect lemma was asserted).

To explore the possible concrete assertions the testing system applies symbolic variables until an expression of non-function type has been reached. Non-function inputs can be explored by enumerating constructors, but higher order functions are more difficult to explore (especially in the presence of non-termination). To handle functional inputs we allow arbitrary assignment of evaluations of functions at symbolic inputs as long as for every evaluation assignment with different outputs there is an assigment that observes a difference in input. This ensures that we allow any observable function to be explored while not needing to enumerate function term syntax.

For example,

$$\lambda f : \mathbb{N} \to \mathbb{B}.\lambda x : \mathbb{N}.\textbf{\textit{if}}\ f\ x\ \textbf{\textit{then}}\ \texttt{assert}[\mathbb{N}]\ \textbf{\textit{else}}\ \texttt{0}$$

the $\texttt{assert}[\mathbb{N}]$ is reachable with $f x \equiv \texttt{True}$ for any instantiation of x.

$$\lambda f : \mathbb{N} \to \mathbb{B}.\lambda x : \mathbb{N}.$$
$$\textbf{\textit{if}}\ f\ x\ \textbf{\textit{then}}\ (\textbf{\textit{if}}\ f\ 3\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ \texttt{assert}[\mathbb{N}])\ \textbf{\textit{else}}\ \texttt{0}$$

can be explored with $f x \equiv \texttt{True}$, $f\ 3 \equiv \texttt{False}$ for any instantiation of x, $x \neq 3$.

This reasoning extends to higher order and recursive functions,

$$\lambda f : (\mathbb{N} \to \mathbb{N}) \to \mathbb{B}.$$
$$\textbf{\textit{if}}\ f\ (\lambda x.x + 0)$$
$$\textbf{\textit{then}}\ \textbf{\textit{if}}\ f\ (\lambda x.0 + x)\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ \texttt{assert}[\mathbb{N}]$$
$$\textbf{\textit{else}}\ \texttt{0}$$

will not explore the $\texttt{assert}[\mathbb{N}]$ since $f\ (\lambda x.x + 0) \equiv \texttt{True}$, $f\ (\lambda x.0 + x) \equiv \texttt{False}$ and there is no instantiation of x, such that $x + 0 \neq 0 + x$.

Finally this exploration can handle non-termination

$$\lambda f : (\mathbb{N} \to \mathbb{N}) \to \mathbb{B}.$$
$$\textbf{\textit{if}}\ f\ \texttt{loopForEver}$$
$$\textbf{\textit{then}}\ \texttt{assert}[\mathbb{N}]$$
$$\textbf{\textit{else}}\ \texttt{0}$$

will explore the $\texttt{assert}[\mathbb{N}]$ since $f\ \texttt{loopForEver} \equiv \texttt{True}$, and no instantiation is needed.

$$\lambda f : (\mathbb{N} \to \mathbb{N}) \to \mathbb{B}.$$
$$\textbf{\textit{if}}\ f\ \texttt{loopForEver}$$
$$\textbf{\textit{thenif}}\ f\ (\lambda x.x)\ \textbf{\textit{then}}\ 1\ \textbf{\textit{else}}\ \texttt{assert}[\mathbb{N}]$$
$$\textbf{\textit{else}}\ \texttt{0}$$

will not explore the $\texttt{assert}[\mathbb{N}]$ since $f\ \texttt{loopForEver} \equiv \texttt{True}$ and $f\ (\lambda x.x) \equiv \texttt{False}$, and no instantiation will be able to differentiate between `loopForEver` and $(\lambda x.x)$ in finite time.

Inspiration from an independent testing phase comes from the practice of Continuous Integration that runs a testing phase that is distinct from compile-time and run-time.

4 Ongoing and Future Work

- Our implementation should be improved to support more standard dependently typed features. Currently the only built in types are dependent functions, natural numbers, equality(Identity), and dependent pairs, and there is only a single type universe. We plan to add support for user defined data types and a more practical universe structure. Ideally the system could also provide feedback on which functions make use of unsafe constructs like `assert` and unbounded recursion so pure functions can be easily recognized as theorems.
- It is currently unclear how to handle parametric features like polymorphic types and modules since parametric reasoning is questionable in the presence of `assert`. For instance, in the empty context $\Pi A : \textbf{\textit{type}}.A \to A$ may not be the identity, since the term $\vdash \lambda A.\lambda - .\texttt{assert}[A] : \Pi A : \textbf{\textit{type}}.A \to A$ would have that type.
- We are currently exploring ideas from Constraint Logic Programming[Jaffar and Lassez 1987; Jaffar and Maher 1994] to make the exploration phase more efficient. Ideally the exploration system could notify the programmer about hard to solve constriant configurations so the programmer could provide solvers for their user defined types and functions.
- As currently implemented, the `assert` construct will choose an arbitrary inhabitant of the type. However, it would be nice to choose the most convenient inhabitant of the type. For instance, $\textbf{\textit{let}}\ a = \texttt{assert}[\mathbb{N}]\ \textbf{\textit{in}}\ \textbf{\textit{let}}\ b = \texttt{fst}\ (\texttt{assert}[\Sigma n : \mathbb{N}.n < a])\ \textbf{\textit{in}}...$ Currently we choose $a = 0$ and b is unsatisfiable. It would be better to choose $a = 1$, and $b = 0$. Unfortunately, the global nature of such assignments is harder to work with. There is a Monadic presentation of nondeterminism that works over non-dependent types (See the Omega Monad[1]). There may be an extention of this theory that works over dependent types. Though mixing dependent types and effects is difficult, there has been interesting recent work in this area[Pédrot and Tabareau 2020].

References

Koen Claessen and John Hughes. 2011. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices* 46, 4 (2011), 53–64.

Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. 2014. QuickChick: Property-based testing for Coq. In *The Coq Workshop*.

Qiao Haiyan. 2003. *Testing and Proving in Dependent Type Theory.* Chalmers University of Technology.

Joxan Jaffar and J-L Lassez. 1987. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* 111–119.

[1]http://hackage.haskell.org/package/control-monad-omega-0.3.2/docs/Control-Monad-Omega.html

Joxan Jaffar and Michael J Maher. 1994. Constraint logic programming: A survey. *The journal of logic programming* 19 (1994), 503–581.

Limin Jia, Jianzhou Zhao, Vilhelm Sjöberg, and Stephanie Weirich. 2010. Dependent types and program equivalence. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 275–286.

Pierre-Marie Pédrot and Nicolas Tabareau. 2020. The Fire Triangle How to Mix Substitution, Dependent Elimination, and Effects.

Vilhelm Sjöberg, Chris Casinghino, Ki Yung Ahn, Nathan Collins, Harley D Eades III, Peng Fu, Garrin Kimmell, Tim Sheard, Aaron Stump, and Stephanie Weirich. 2012. Irrelevance, heterogeneous equality, and call-by-value dependent type systems. *arXiv preprint arXiv:1202.2923* (2012).