Extracting and evolving the Idris core type theory

Justus Matthiesen
mail@justusmatthiesen.com
University of Edinburgh

Idris [2] is an implementation of a dependently-typed programming language. Checking a program written in its user-friendly surface language involves elaboration to a lower level core type theory. This core type theory is currently being re-engineered[1] by Edwin Brady and now features explicit quantity annotations, allowing variables to be marked as erasable, linear or unrestricted, as well as *case trees*, first described in [6] and later given formal treatment in [4], which serve as a more convenient elaboration target for functions defined by dependent pattern matching [5]. A self-standing type checker for the core itself is also being developed, which will eventually allow re-checking of elaborated programs.

In this talk, we report on a parallel work-in-progress effort to extract a set of typing rules[2] for this core type theory from the Idris code base. The hope is that this project will aid bug finding, enable meta-theoretic investigations, help collaborators work on Idris-related projects without having to delve into the code base, and serve as a basis to explore extensions to the core type theory.

We can already report some early successes: documenting Idris' quantity system helped us uncover and fix a number of bugs in the linearity checker and having a concise language description is currently proving to be a useful design tool in making case trees more expressive by allowing case splitting on arbitrary expressions with the end goal of making dependent pattern matching a first-class citizen.

1 Quantity annotations

Inspired by Quantitative Type Theory [1], Idris' type system uses quantity annotations [3] to indicate that a variable in context maybe be erased (0), must be used exactly once (1), or can be used arbitrarily (ω) at runtime. Quantities form a semiring with order $0 < 1 < \omega$.

More formally, the typing judgement[3] $\Sigma; \Gamma \vdash t :^{p} A$ amounts to checking that term t has type A in context Γ at *ambient* quantity p with definitions Σ. The ambient quantity behaves much like QTT's erased/present flag but it also acts as cost multiplier for variable accesses: suppose we want to infer the type of an application $f \cdot^{\omega} t$ at ambient quantity 1 and already know that $\Sigma; \Gamma \vdash f :^{1} (x :^{\omega} A) \to B$. We now not only need to check that argument t has type A but also that we are allowed to make ω copies of t. One way to achieve this would be to restrict the context to those variables that support ω copies, $\Sigma; \Gamma \setminus \omega \vdash t :^{1} A$, where $p \setminus q$ is the restriction operation making p-annotated variables that do not support at least q copies unavailable at runtime

$$p \setminus q = \begin{cases} p & \text{if } q \leq p \\ 0 & \text{otherwise} \end{cases}$$

Idris, however, uses a variable rule that requires any accessed variable to have quantity larger or equal to the ambient quantity

$$\frac{(x :^{q} A) \in \Gamma \qquad p \leq q}{\Sigma; \Gamma \vdash x :^{p} A} \text{ [Var]}$$

allowing us to use $\Sigma; \Gamma \vdash t :^{\omega} A$ instead. In fact, we should always have

$$\Sigma; \Gamma \vdash t :^{s \cdot p} A \iff \Sigma; \Gamma \setminus s \vdash t :^{p} A$$

This limited form of subusaging is departure from Atkey's presentation of QTT, where usage annotations must always be precise.

2 Dependent pattern matching

Instead of elaborating functions defined via pattern matching into a core calculus featuring only eliminators, Idris stops at case trees as a more natural intermediate language. Case trees are restricted to splitting only on variables but this simplicity comes at a cost: case trees do not support a substitution operation and therefore also have no equational theory. As a further consequence, case expressions on the right-hand side and *with*–abstraction [7] have to be elaborated to top-level functions, obscuring the structure of the original program and requiring special treatment in some parts of the code base, e.g. the termination checker.

To alleviate the issue, we equip each case expression with a suspended substitution, allowing abstraction before case splitting and preventing case expressions from commuting arbitrarily with substitutions.

3 Future work

We plan to use the presented type system to tackle a variety of projects, including meta-theoretic correctness results for erasure and linearity annotations, strengthening the equational theory of pattern matching, exploring primitives for sparsely manipulating the context and managing dependencies, and bringing observational equality to Idris.

[1] The next iteration is being developed at https://github.com/edwinb/Yaffle.
[2] The typing rules are maintained at https://github.com/mjustus/idris-core.
[3] We follow Idris in splitting type checking and linearity into two separate judgements. Type checking only enforces quantity annotations locally but does not enforce that linear variables are used precisely once globally.

References

[1] Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 56–65. https://doi.org/10.1145/3209108.3209189

[2] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

[3] Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming (ECOOP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:26. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

[4] Jesper Cockx and Andreas Abel. 2020. Elaborating dependent (co)pattern matching: No pattern left behind. *Journal of Functional Programming* 30 (2020), e2. https://doi.org/10.1017/S0956796819000182

[5] Thierry Coquand. 1992. Pattern Matching with Dependent Types. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*. Båstad. https://www.cse.chalmers.se/~coquand/pattern.ps

[6] Healfdene Goguen, Conor McBride, and James McKinna. 2006. Eliminating Dependent Pattern Matching. In *Algebra, Meaning, and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 521–540. https://doi.org/10.1007/11780274_27

[7] Conor McBride and James McKinna. 2004. The view from the left. *Journal of Functional Programming* 14, 1 (2004), 69–111. https://doi.org/10.1017/S0956796803004829