The Honors College at the University of Missouri-Kansas City

Performance Between Universes: Type-Informed Optimisation for Structural Abstraction and Environment-Specific Data

Samuel Lim

May 2020

Written under the direction of Professor Sejun Song

Department of Computer Science Electrical Engineering

School of Computing and Engineering

A thesis submitted in partial fulfillment of the requirements to graduate as an Honors Scholar from the University of Missouri-Kansas City

Abstract

We propose an implementation process for type systems to permit subdivision of types into distinct universes based on individual type theories each presenting their own unique properties. Multi-universe systems can interface between smooth and discrete definitions of similar values at the type level. We then present a potential resolution framework for these values to be incrementally optimised by a preprocessor for both self-defined primitive types and high-dimensional abstractions.

Background

Innovations in modern understandings of type theory alongside classical logical and categorical frameworks (see Martin-Löf) have been foundational to the growth of programming languages in recent years [2], even leading up to the solution of long-standing open problems in the programming language community [4]. Applied operational semantics in many compiled and interpreted languages with said formalisation have seen acceptance into mainstream and developing ecosystems indeed. Languages like F# within the .NET ecosystem, Scala 3 (across both the JVM and the Dotty compiler)[1], and more recent developments like those seen in Idris 2 present features specific to each locale, such as unit-of-measurement type definitions (see Kennedy) and pure I/O mocking [6] [8]. Programming languages that embed logic within affine types such as Rust[2] add critical value by bringing exposure of value-agnostic properties (see

[1]See https://github.com/lampepfl/dotty for current information of the compatible features between the Dotty compiler and JVM feature integration

[2]https://rust-lang.org/

def. lifetime[3]) to the developer at face value [11]. We seek a process to visibly represent environment-specific features to language users while allowing optimisation techniques across individual type domains in order to better understand the scale of abstraction between languages and compilation environments and how well optimisation techniques process foreign information without extraneous passes.

Inter-universality

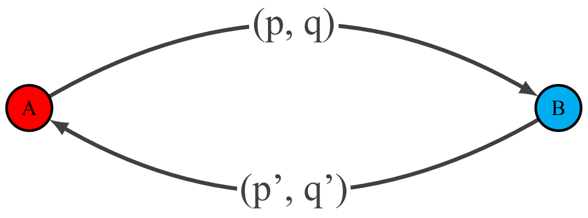

Figure I. *Information exchange between type universes.*

To best understand an approach to unifying type systems for language users, we first establish definitions for common terminology set in place to disambiguate motivations of type theory, category theory, information theory, compiler development, and application domains. With respect to usage in programming languages, a data **type** T is an internal representation of a set of properties about an object or data with an instance v (written $v : T$). These properties, or **metadata**, hold information such as **type attributes**, a method of specifying typed data within type T, **type signatures** denoting the literal syntax and declaration of types, **supertypes**, the types from which T derives and of which it assumes all properties (written $T <: S$ where S is a supertype of T)[4], **type invariants** where all values or instances of T may assume as valid while remaining of type T. We then present a notion of a **type universe** U, with respect to interacting languages, denoting the set of all possible types under a specific compiler, language or subset of a language, with a set of global invariants and metadata P that formalise a type’s construction, structure, and destructuring should it be passed to another context.

Type universes, in particular, become immediately relevant when discussing non-equivalent type systems of languages. Oftentimes the use-case may serve as a formalisation for questions of language interoperability, or the ability to convey information from one computational language

[3]https://doc.rust-lang.org/rust-by-example/scope/lifetime.html

[4]Type T may also be considered a subtype of S

to another and vica-versa. Largely due to the invariants imposed from one language or another, interoperability can be represented as a process of information exchange. As visualised in Figure I, a type universe A from a language L_A may convey information with types

$$S_T^V = \{v \in V | \text{ let } v : T\}$$

to a type universe B from a language L_B transfers information by gaining type information p and stripping type metadata q. Information can be relayed from B to A similarly with a process function emitting gained and lost metadata p' and q' respectively. Previous work in Girard's linear logic and subsequently linear (and affine) type systems have demonstrated an ability to preserve such properties and metadata of a value for the duration of its usage as maintained within an enclosed type system [3]. However, this has little influence over preservation of metadata when moving across universes, as would be later seen in many cases.

In application, the difference of the two cases appears when interacting between two or more languages with separate operational semantics. Type boundaries can be seen directly when working with C under a foreign-function interface (or FFI) [5]. A language seeking to call into code written in C may choose to strip all metadata and formal properties of information as it exits the language before entering C as typeless information, or it may choose to subsume all primitives of the concerned type system (in this case, C's type system) into its own. Some languages may commonly allow both paradigms within its own semantics [5]. By implication, this consequently implies

$$preserves(S_A^V, S_B^V) \not\Rightarrow preserves(S_B^{V'}, S_A^{V'})$$

since a type universe B may subsume A with the direct understanding that strictly $A \subset B$.

Implicit Semantics

The applications of introducing inter-universal definitions are three-fold: 1) General-purpose languages may interface with domain-specific universes unimpeded as to enable new programming paradigms, 2) General-purpose languages may extend their functionality by simulating an anonymous language with new semantics and integrating it into it own type system under a second universe, and 3) languages may optimise pre-existing definitions and functionality by reduction and lowering into optimised domain-specific universes within a formal subset of said language.

Implicit semantics may present themselves in cases of direct domain application. Rich theories of type-level formalisation have been made available for areas like probability, both in theory (see Cooper) and interactive programming languages [7]. Early attempts at compiler-level reduction have previously taken on forms, as seen in units of measure data attributes [1], which are both available as compiler-reducible types in F# or user-defined compile-time attributes, as seen in Rust. A more recent display of interoperable type universes can be found in the Stan programming language, where the language has a clear reduction model to compiler-intrinsic probabilistic functions [9].

As a methodological proposition, metadata-retaining reduction processes have come become popular more recently. Idris 2 manages its own type universe while integrating the entirety of its semantics when compiling to Chez Scheme[5]. The Futhark programming language also presents structural typing and copy elision as presently abtracted in the language over multiple GPU compute targets [10]. Similarly, multi-universe definitions can be targeted as language-level compiler extensions to a present language lowering to an adjoint proof system hoisted to a compiler back-end like LLVM. A predecessor implementation permitting type generics can be seen in current use with the Rust programming language[6].

Conclusions and Future Work

We have laid out a simple lexicon of common terminology for applied type theory with respect to computational languages and optimisation, and we have introduced the notion of a type universe for further use in crossing multiple languages and domains without unnecessary type conflict. Proposed definitions can be of use in type resolution, interoperability, optimisation, and productivity in domain applications.

To further the approach of an inter-universal language paradigm, we take our next steps in approaching formal definitions in proof and multi-paradigm languages like Agda and Idris

[5]https://www.type-driven.org.uk/edwinb/why-is-idris-2-so-much-faster-than-idris-1.html

[6]https://github.com/rust-lang/chalk

(respectively) [7] for the same formulations.

Acknowledgements

We thank members of the Rust and Julia programming language communities as well as those active in the Category Theory Café for their continued input and mindful direction during the writing of this paper. We also would like to thank contributors to the nCatLab project for their helpful introduction into the subject material to be presented as the foundation of this work, including those who have previously written towards PLDI.

References

[1] N. Gehani, "Units of measure as a data attribute", *Computer Languages*, vol. 2, no. 3, pp. 93–111, 1977.

[2] P. Martin-Löf and G. Sambin, *Intuitionistic type theory*. Bibliopolis Naples, 1984, vol. 9.

[3] P. Wadler, "Linear Types Can Change the World!", in *PROGRAMMING CONCEPTS AND METHODS*, North, 1990.

[4] T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott, "Normalization by evaluation for typed lambda calculus with coproducts", in *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, IEEE, 2001, pp. 303–310.

[5] M. Furr and J. S. Foster, "Checking type safety of foreign function calls", *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 62–72, 2005.

[6] A. Kennedy, "Types for units-of-measure: Theory and practice", in *Central European Functional Programming School*, Springer, 2009, pp. 268–305.

[7] R. Cooper, S. Dobnik, S. Lappin, and S. Larsson, "A probabilistic rich type theory for semantic interpretation", in *Proceedings of the EACL 2014 Workshop on Type Theory and Natural Language Semantics (TTNLS)*, 2014, pp. 72–79.

[8] E. Brady, *Type-driven development with Idris*. Manning Publications Company, 2017.

[7] https://github.com/amadeusine/iutt-formal

[9] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell, “Stan: A probabilistic programming language”, *Journal of statistical software*, vol. 76, no. 1, 2017.

[10] T. Henriksen, N. G. Serup, M. Elsman, F. Henglein, and C. E. Oancea, “Futhark: purely functional GPU-programming with nested parallelism and in-place array updates”, in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 556–571.

[11] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, “RustBelt: Securing the foundations of the Rust programming language”, *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–34, 2017.