Unboxed Dependent Types

Constantine Theocharis[1]

University of St Andrews
kt81@st-andrews.ac.uk

When the reduction rules of type theory become part of the static semantics, it is impossible to always compute the memory layout of a type during compilation. This happens when polymorphism is present but monomorphisation is not, and in particular when dependent types are part of the mix, since the dependency might be on runtime values.

In this ongoing work, we formulate a dependent type theory where sizes of types are always known at compile-time, and thus compilation can target a language like C. Indirection is opt-in and can be avoided, leading to efficient code that enables cache locality. This is done by *indexing* syntactic types by a metatheoretic type describing memory layouts: Bytes. This leads to a notion of representation polymorphism. Unboxed data in functional languages has been explored before [6, 5], but results in overly restricted polymorphism or complex theories with multiple levels of kinds separating values and computations, and without support for full dependent types. Our approach is lightweight and extends to full dependent types.

The staging view of *two-level type theory* (2LTT) [4] has been explored by Kovács in the general setting [7] as well as in the setting of closure-free functional programming [8]. Inspired by a note in the afforementioned works, we can embed our unboxed type theory as the object language of a 2LTT, which allows us to write type-safe metaprograms that compute representation-specific constructions. For example, we can formulate a universe of flat protocol specifications in the style of Allais [2], and interpret it in the unboxed object theory. We needn't compromise on the usage of dependent types either; as opposed to [8], the object theory is itself dependently typed and thus we can encode unboxed higher-order polymorphic functions as part of the final program, as long as all *sizes* (but not necessarily all types) are known after staging.

Basic setup We formulate our system as an MLTT-style type theory, using the intrinsic QIIT syntax of Altenkirch and Kaposi [3]. We assume access to a metatheoretic type of Bytes with

$$0, 1 : \mathsf{Bytes} \qquad + : \mathsf{Bytes} \to \mathsf{Bytes} \to \mathsf{Bytes} \qquad \mathsf{ptr} : \mathsf{Bytes} \qquad \times : \mathsf{Nat} \to \mathsf{Bytes} \to \mathsf{Bytes}\,.$$

The constant ptr defines the size of a pointer. Any model of the signature above will suffice; such a model might encode a sophisticated layout algorithm with padding, for example.

First, types are indexed not just by contexts, but also by bytes:[1] $\mathsf{Ty} : \mathsf{Con} \to \mathsf{Bytes} \to \mathbf{Set}$. We have the following basic type and term formers:

$$\begin{array}{lll} \mathcal{U}_{\_} : \mathsf{Bytes} \to \mathsf{Ty}\ \Gamma\ 0 & \mathsf{El} : \mathsf{Tm}\ \Gamma\ \mathcal{U}_b \simeq \mathsf{Ty}\ \Gamma\ b : \mathsf{code} & \langle - \rangle : \mathsf{Tm}\ \Gamma\ \mathcal{U}_b \to \mathsf{Tm}\ \Gamma\ \mathcal{U} \\ \mathcal{U} : \mathsf{Ty}\ \Gamma\ 0 & \mathsf{El}_{\square} : \mathsf{Tm}\ \Gamma\ \mathcal{U} \to \mathsf{Ty}\ \Gamma\ \mathsf{ptr} & \mathsf{box} : \mathsf{Tm}\ \Gamma\ A \simeq \mathsf{Tm}\ \Gamma\ (\mathsf{El}_{\square}\ \langle \mathsf{code}\ A \rangle) : \mathsf{unbox} \end{array}$$

The type $\mathcal{U}$ is the type of codes for types of an unknown size, while the $\mathcal{U}_b$ type is the type of codes for types of size b. The El interpretation maps codes for types of size b to actual types of size b, while the $\mathsf{El}_{\square}$ interpretation maps codes for any type to actual types which *box* their contents. In other words, internally, if $A : \mathcal{U}$, then $\mathsf{El}_{\square}\ A$ can be used as a type and at runtime the data of A will be under a heap-allocated pointer indirection. On the other hand, if $B : \mathcal{U}_b$, then $\mathsf{El}\ B$ can be used as a type and at runtime the data of B will be stored *inline*, since it

[1] For brevity we will not regard issues of universe sizing, but this can be accomodated without issue.

is known that B takes up exactly b bytes. Codes for types of any kind take up no space at runtime because they are erased. Additionally, for any code t in $\mathcal{U}_b$, we can get a code $\langle t \rangle$ in $\mathcal{U}$ by 'forgetting' that the size of t is b. Finally, we have boxing and unboxing operators for types of a known size. Contexts Γ store the size of each type, such that $|\Gamma| : \mathsf{Bytes}$ is the sum of the sizes of its types. The action of substitutions on types does not vary their sizes: $-[-] : \mathsf{Ty}\ \Gamma\ b \to \mathsf{Sub}\ \Delta\ \Gamma \to \mathsf{Ty}\ \Delta\ b$.

Unboxed pairs, boxed and unboxed closures This setup can be augmented with Π and Σ types, where the dependency is *uniform* with respect to layout:

$$\begin{aligned}
\Pi_\square &: (A : \mathsf{Ty}\ \Gamma\ a) \to \mathsf{Ty}\ (\Gamma \triangleright A)\ b \to \mathsf{Ty}\ \Gamma\ (\mathsf{ptr} + \mathsf{ptr}) \\
\Pi_k &: (A : \mathsf{Ty}\ \Gamma\ a) \to \mathsf{Ty}\ (\Gamma \triangleright A)\ b \to \mathsf{Ty}\ \Gamma\ (k + \mathsf{ptr}) \\
\Sigma &: (A : \mathsf{Ty}\ \Gamma\ a) \to \mathsf{Ty}\ (\Gamma \triangleright A)\ b \to \mathsf{Ty}\ \Gamma\ (a + b)
\end{aligned}$$

For functions, we have a choice of whether to box the captures or not. In the latter case, we must annotate them with the size of their captures. Conversely, pairs are stored inline; their size is the sum of the sizes of their components. The term formers remain mostly unchanged; the only new case is Π_k whose lambda terms declare their captures through a substitution: $\lambda : (\rho : \mathsf{Sub}\ \Gamma\ \Delta) \to \mathsf{Tm}\ (\Delta \triangleright A)\ B \to \mathsf{Tm}\ \Gamma\ (\Pi_{|\Delta|}\ A\ B)[\rho]$. In practice this substitution can be inferred from the surface program.

First-class byte values with staging This type theory can be embedded as an object language $\mathbb{O}$ of a 2LTT. On the meta level, we have a type former $\mathbb{b} : \mathsf{Ty}_1\ \Gamma$ of byte values, and term formers that mirror Bytes.[2] In an empty context, in a theory with canonicity, we get an evaluation function $\mathsf{ev} : \mathsf{Tm}_1\ \cdot\ \mathbb{b} \to \mathsf{Bytes}$. Adding Π types to the meta language allows abstraction over byte values. Moreover, the meta level has a type former $\Uparrow_b : \mathsf{Ty}_0\ \Gamma\ b \to \mathsf{Ty}_1\ \Gamma$ for embedding $(b : \mathsf{Tm}\ \Gamma\ \mathbb{b})$-sized types from the object fragment. If the final program is of the form $p : \mathsf{Tm}_1\ \cdot\ (\Uparrow_k A)$, after staging we get an object term of size $\mathsf{ev}\ k$.

Example: Maybe as a tagged union Let's take a look at how to define the Maybe type internally in such a way that its data is stored contiguously as a tagged union without indirections.[3]. We assume access to a type $\mathsf{Pad}\ b : \mathsf{Ty}\ \Gamma\ b$ which is the unit type that takes up b bytes, with sole constructor pad akin to tt.

$$\begin{aligned}
&\mathsf{Maybe}_b : \Pi_0\ (T : \mathcal{U}_b)\ \mathcal{U}_{1+b} \\
&\mathsf{Maybe}_b = \lambda\ T.\ \Sigma\ (x : \mathbb{2})\ (\mathsf{if}\ x\ \mathsf{then}\ T\ \mathsf{else}\ (\mathsf{Pad}\ b))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{nothing}_b : \Pi_0\ (T : \mathcal{U}_b)\ (\mathsf{Maybe}_b\ T) &\qquad& \mathsf{just}_b : \Pi_0\ (T : \mathcal{U}_b)\ \Pi_0\ T\ (\mathsf{Maybe}_b\ T) \\
&\mathsf{nothing}_b = \lambda\ _.\ (\mathsf{false}, \mathsf{pad}) && \mathsf{just}_b = \lambda\ _\ t.\ (\mathsf{true}, t)
\end{aligned}$$

Example: Arrays We can extend the language with flat arrays of a statically known size $n : \mathsf{Nat}$ by $\mathsf{Array}\ n : \mathsf{Ty}\ \Gamma\ b \to \mathsf{Ty}\ \Gamma\ (n \times b)$, as well as of a runtime size $r : \mathsf{Tm}\ \Gamma\ \mathbb{N}$ by $\mathsf{DynArray}\ r : \mathsf{Ty}\ \Gamma\ b \to \mathsf{Tm}\ \Gamma\ \mathcal{U}$ whose inhabitants can only be accessed under a box. For example,

$$\Sigma\ (n : \mathbb{N})\ (\mathsf{El}_\square\ (\mathsf{DynArray}\ n\ (\mathsf{Array}\ 2\ (\Pi_k\ A\ B))))$$

is the type of dynamically-sized arrays of pairs of closures from A to B of capture size k.

[2]More precisely, a free extension [9] of Bytes by the meta-level syntax.
[3]This is similar to the approach of languages such as Rust [1].

References

[1] std::option - Rust. https://doc.rust-lang.org/std/option/. Accessed: 2025-3-7.

[2] Guillaume Allais. Seamless, correct, and generic programming over serialised data. *arXiv [cs.PL]*, 20 October 2023.

[3] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 18–29, New York, NY, USA, 11 January 2016. Association for Computing Machinery.

[4] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications. *Math. Struct. Comput. Sci.*, 33(8):688–743, September 2023.

[5] Paul Downen. Call-by-unboxed-value. *Proc. ACM Program. Lang.*, 8(ICFP):845–879, 21 August 2024.

[6] Simon L Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture*, Lecture notes in computer science, pages 636–666. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991.

[7] András Kovács. Staged compilation with two-level type theory. *Proc. ACM Program. Lang.*, 6(ICFP):540–569, 29 August 2022.

[8] András Kovács. Closure-free functional programming in a two-level type theory. *Proc. ACM Program. Lang.*, 8(ICFP):659–692, 15 August 2024.

[9] Jeremy Yallop, Tamara von Glehn, and Ohad Kammar. Partially-static data as free extension of algebras. *Proc. ACM Program. Lang.*, 2(ICFP):1–30, 30 July 2018.