Unboxed Dependent Types

Constantine Theocharis and Ellis Kesterton

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

When the reduction rules of type theory become part of its static semantics, it is sometimes impossible to compute the memory size in bytes 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 the memory size of a type is always known at compile-time, and thus compilation can directly target a low-level language. Boxing 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 [7, 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 dependent types.

The staging view of *two-level type theory* (2LTT) [3] has been explored by Kovács in the general setting [9] as well as in the setting of closure-free functional programming [10]. Inspired by a note in the aforementioned 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 [10], the object theory is itself dependently typed and thus we can encode unboxed higher-order polymorphic functions as part of the final program, because all *sizes* (not necessarily all types) are known after staging.

Basic setup We formulate our system in the form of a second-order generalised algebraic theory (SOGAT) [8]. We assume 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.

To begin with, types are indexed by bytes:[1] $\mathsf{Ty} : \mathsf{Bytes} \to \mathbf{Set}$. We have the following basic type and term formers:

$$\begin{aligned} &\mathsf{U}_{\_} : \mathsf{Bytes} \to \mathsf{Ty}\ 0 && \mathsf{Tm}\ \mathsf{U}_b = \mathsf{Ty}\ b \\ &\square : \mathsf{Ty}\ b \to \mathsf{Ty}\ \mathsf{ptr} && \mathsf{box} : \mathsf{Tm}\ \square A \simeq \mathsf{Tm}\ A : \mathsf{unbox}\,. \end{aligned}$$

The universe U_b describes types whose inhabitants take up b bytes, with a Grothendiek-style identification $\mathsf{Tm}\ \mathsf{U}_b = \mathsf{Ty}\ b$. The $\square$ type former takes sized types to types which *box* their contents. In other words, for a type A of some size a, an inhabitant of data of $\square A$ will be stored on the heap behind a pointer indirection. On the other hand, an inhabitant of A will be stored *inline* on the stack, since it is known that A takes up exactly a bytes. Codes for types of any kind take up no space at runtime because they are erased. Additionally, we have

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

boxing and unboxing operators for types of a known size. Since we present this as a second-order theory, we can mechanically derive its first-order presentation [8], with explicit contexts. There, contexts Γ record the size of each type, such that $\mathrm{b}(\Gamma) : \mathsf{Bytes}$ can be defined as the sum of the sizes of its types. The action of substitutions on types does not vary their sizes: $-[-] : \mathsf{Ty}\ \Gamma\ b \to \mathsf{Sub}\ \Delta\ \Gamma \to \mathsf{Ty}\ \Delta\ b$.

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

$$\begin{aligned}\Pi &: (A : \mathsf{Ty}\ a) \to (\mathsf{Tm}\ A \to \mathsf{Ty}\ b) \to \mathsf{Ty}\ \mathsf{ptr}\\ \Sigma &: (A : \mathsf{Ty}\ a) \to (\mathsf{Tm}\ A \to \mathsf{Ty}\ b) \to \mathsf{Ty}\ (a + b)\,.\end{aligned}$$

For functions, we store a pointer to an allocation containing both code and data. Alternatively, we could separate the two by sizing functions as $\mathsf{ptr} + \mathsf{ptr}$. Conversely, pairs are stored inline; their size is the sum of the sizes of their components. The term formers remain unchanged.

It could be desirable to have a function type with *unboxed* captures, which might look like

$$\Pi_k : (A : \mathsf{Ty}\ a) \to (\mathsf{Tm}\ A \to \mathsf{Ty}\ b) \to (\mathsf{ptr} + k)\,,$$

annotated with the size of its captures k. This is not expressible as a second-order construct because its term former must know about captured variables: $\lambda : (\rho : \mathsf{Sub}\ \Gamma\ \Delta) \to \mathsf{Tm}\ (\Delta \triangleright A)\ B \to \mathsf{Tm}\ \Gamma\ (\Pi_{\mathrm{b}(\Delta)}\ A\ B)[\rho]$. This also means it is not immediately compatible with 2LTT. We leave this as future work, which would likely involve a modality for closed object terms.

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 $\mathbb{B} : \mathsf{Ty}_1$ of byte values, and term formers that mirror Bytes. In an empty context, in the first-order formulation, we get an evaluation function $\mathsf{ev} : \mathsf{Tm}_1\ \bullet\ \mathbb{B} \to \mathsf{Bytes}$. Adding Π types to the meta language allows abstraction over byte values. Moreover, the meta level has a type former $\Uparrow_b : \mathsf{Ty}_0\ b \to \mathsf{Ty}_1$ for embedding $(B : \mathsf{Tm}_1\ \mathbb{B})$-sized types from the object fragment. If the final program is of the form $p : \mathsf{Tm}_1\ \bullet\ (\Uparrow_k A)$, 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.[2]. We assume access to a type $\mathsf{Pad}\ b : \mathsf{U}_b$ which is the unit type that takes up b bytes, with sole constructor pad akin to tt, and $\mathsf{Bool} : \mathsf{U}_1$ which takes up a single byte:

$$\begin{aligned}&\mathsf{Maybe} : (T : \mathsf{U}_b) \to \mathsf{U}_{1+b}\\ &\mathsf{Maybe}\ T = (x : \mathsf{Bool}) \times \mathsf{if}\ x\ \mathsf{then}\ T\ \mathsf{else}\ \mathsf{Pad}\ b\end{aligned}$$

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

Computational irrelevance and runtime-sized data Annotating object-level types with bytes provides a convenient way to handle computational irrelevance without further modifying the structure of contexts. This is possible through a monadic modality

$$|-| : \mathsf{Ty}\ b \to \mathsf{Ty}\ 0 \qquad \|-\| : \mathsf{Tm}\ A \to \mathsf{Tm}\ |A|$$

[2]This is similar to the approach of languages such as Rust [1].

which is idempotent by $(A : \mathsf{Ty}\ 0) \to |A| = A$, extending to all zero-sized types. It also has an appropriate eliminator form. With this we can reproduce, for example, quantitative type theory instantiated with the $\{0, \omega\}$ semiring [4]. This means that we can now use object-level types which are entirely erased: $\mathsf{reverse} : \{n : |\mathsf{Nat}|\} \to \mathsf{Vect}\ T\ n \to \mathsf{Vect}\ T\ n$.

Additionally, we often want to handle data whose size is *not* known at compile-time, but is known at runtime; most commonly, heap-backed arrays, but also other dynamically-sized flat data structures. This is achievable by indexing the universe U by partially-static [11] byte values. Object-level types $\mathsf{Ty}\ b$ are now identified only with $\mathsf{U}_{\mathsf{sta}\ b}$ where $\mathsf{sta} : \mathsf{Tm}_1\ \mathbb{B} \to \mathsf{Tm}_1\ \mathbb{B}^{\mathsf{PS}}$. We can then add appropriate type formers to the theory for the construction of runtime-sized data such as pairs. Their inhabitants cannot directly be stored on the stack, but they can be constructed and manipulated on the heap. To do this, boxing is relaxed to allow runtime-sized data, and we must have a type for 'generating' unsized data. We plan on presenting examples of this in our presentation.

Formalisation, semantics and implementation We have formalised most parts of the sketched system by a shallow embedding in Agda, including the irrelevance modality. We have also formulated a semantics in terms of an untyped lambda calculus which is nevertheless *sized* just like the system we presented. Crucially, the sizes of all constructs in this target language must be *non-zero*, which forces us to translate away all zero-sized types. This justifies the irrelevance modality as well as the erased codes for types. We are currently working on a proof-of-concept implementation that targets LLVM.

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] 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.

[4] Robert Atkey. Syntax and semantics of quantitative type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 56–65. Association for Computing Machinery.

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

[6] Richard A Eisenberg and Simon Peyton Jones. Levity polymorphism. *SIGPLAN Not.*, 52:525–539.

[7] 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.

[8] Ambrus Kaposi and Szumi Xie. Second-order generalised algebraic theories: Signatures and first-order semantics.

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

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

[11] 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.