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. This has been explored before, but in the context of non-dependent functional languages, and often comes with a heuristic final pass which adds indirection to types whose size cannot be heuristically determined. Our approach requires no heuristics, and extends to full dependent types.

The staging view of *two-level type theory* (2LTT) [3] has been explored by Kovács in the general setting [4] as well as in the setting of closure-free functional programming [5]. Inspired by a note in the afforementioned works, we 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 [1], and interpret it in the unboxed object theory. We needn't compromise on the usage of dependent types either; as opposed to [5], our object theory is dependently typed and thus we can encode higher-order polymorphic functions as part of the final program, but without hiding data behind indirections.

Basic setup We formulate our system as an MLTT-style type theory, using the intrinsic QIIT syntax of Altenkirch and Kaposi [2]. We assume access to a metatheoretic type of Bytes with

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

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 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:

$$\mathcal{U}_{\_} : \mathsf{Bytes} \to \mathsf{Ty}\ \Gamma\ 0 \qquad \mathsf{El} : \mathsf{Tm}\ \Gamma\ \mathcal{U}_b \simeq \mathsf{Ty}\ \Gamma\ b : \mathsf{code} \qquad \langle - \rangle : \mathsf{Tm}\ \Gamma\ \mathcal{U}_b \to \mathsf{Tm}\ \Gamma\ \mathcal{U}$$
$$\mathcal{U} : \mathsf{Ty}\ \Gamma\ 0 \qquad \mathsf{El}_\Box : \mathsf{Tm}\ \Gamma\ \mathcal{U} \to \mathsf{Ty}\ \Gamma\ \mathsf{ptr} \qquad \mathsf{box} : \mathsf{Tm}\ \Gamma\ A \simeq \mathsf{Tm}\ \Gamma\ (\mathsf{El}_\Box\ \langle \mathsf{code}\ A \rangle) : \mathsf{unbox}$$

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 $\mathsf{El}_\Box$ interpretation maps codes for any type to actual types which *box* their contents. In other words, internally, if $A : \mathcal{U}$, then $\mathsf{El}_\Box\ A$ 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 we know the size of t is b. Finally, we have boxing and unboxing operators for types of a known size.

Contexts and substitutions Contexts Γ store the size of each type, such that $|\Gamma| : \mathsf{Bytes}$ is the sum of the sizes of its types: Extension is $\triangleright : (\Gamma : \mathsf{Con}) \to \{b : \mathsf{Bytes}\} \to \mathsf{Ty}\ \Gamma\ b \to \mathsf{Con}$. Substitutions are defined as usual; we do not really need to augment the standard definition other than adding implicit arguments for bytes. However, we must ensure that the action of substitutions on types does not vary their sizes: $-[-] : \mathsf{Ty}\ \Gamma\ b \to \mathsf{Sub}\ \Delta\ \Gamma \to \mathsf{Ty}\ \Delta\ b$.

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

$$\Pi : (A : \mathsf{Ty}\ \Gamma\ a) \to \mathsf{Ty}\ (\Gamma \triangleright A)\ b \to \mathsf{Ty}\ \Gamma\ \mathsf{ptr} \qquad \Sigma : (A : \mathsf{Ty}\ \Gamma\ a) \to \mathsf{Ty}\ (\Gamma \triangleright A)\ b \to \mathsf{Ty}\ \Gamma\ (a + b)$$

A function's size in memory is just a pointer to its closure allocation which should also contain the code pointer. For now we do not handle unboxed closure captures, but the inputs and outputs are unboxed. Conversely, pairs are stored inline; their size is the sum of the sizes of their components.

Padding type and unit Sometimes we want to store some extra bytes to pad a smaller type in order to reach a desired size. For this reason we add an extra type former

$$\mathsf{Pad} : (b : \mathsf{Bytes}) \to \mathsf{Ty}\ \Gamma\ b \qquad \mathsf{pad} : \mathsf{Tm}\ \Gamma\ (\mathsf{Pad}\ b) \qquad \mathsf{pad}\text{-}\eta : (t : \mathsf{Tm}\ \Gamma\ (\mathsf{Pad}\ b)) \to t = \mathsf{pad}$$

which is a generalisation of the unit type that takes up b bytes in memory. We can then define $\mathbb{1} := \mathsf{Pad}\ 0$ and $\mathsf{tt} := \mathsf{pad}$.

First-class layouts with staging

0.1 Example: Maybe as a tagged union

First, let's take a look at how to define the Maybe type in such a way that its data is stored contiguously as a tagged union. This is similar to how languages such as Rust store it.

$$\begin{aligned}
&\mathsf{Maybe}_b : \Pi\ (T : \mathcal{U}_b)\ \mathcal{U}_{b+1} \\
&\mathsf{Maybe}_b = \lambda\ T.\ \Sigma\ (x : 2)\ (\mathsf{if}\ (x'.\ \mathcal{U}_b)\ x\ T\ (\mathsf{Pad}\ b)) \\
\\
&\mathsf{nothing}_b : \mathsf{Maybe}_b\ T \\
&\mathsf{nothing}_b = (\mathsf{false}, \mathsf{pad}) \\
\\
&\mathsf{just}_b : T \to \mathsf{Maybe}_b\ T \\
&\mathsf{just}_b = \lambda\ t.\ (\mathsf{true}, t) \\
\\
&\mathsf{maybe}_b : \{T : \mathcal{U}_b\} \to (E : \mathsf{Maybe}_b\ T \to \mathcal{U}_c) \\
&\to E\ \mathsf{nothing} \to ((t : T) \to E\ (\mathsf{just}\ t)) \\
&\to (m : \mathsf{Maybe}_b\ T) \to E\ m \\
&\mathsf{maybe}_b = \lambda\ E\ n\ j\ m.\ \ldots
\end{aligned}$$

References

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

[2] 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.

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

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