Double-Ended Bit-Stealing for Algebraic Data Types

MARTIN ELSMAN, University of Copenhagen, Denmark

Algebraic data types are central to the development and evaluation of most functional programs. It is therefore important for compilers to choose compact and efficient representations of such types, in particular to achieve good memory footprints for applications.

Algebraic data types are most often represented using blocks of memory where the first word is used as a so-called tag, carrying information about the constructor, and the following words are used for carrying the constructor's arguments. As an optimisation, lists are usually represented more compactly, using a technique called bit-stealing, which, in its simplest form, uses the word-alignment property of pointers to byte-addressed allocated memory to discriminate between the nil constructor (often represented as 0x1) and the cons constructor (aligned pointer to allocated pair). Because the representation supports that all values can be held uniformly in one machine word, possibly pointing to blocks of memory, type erasure is upheld.

However, on today's 64-bit architectures, memory addresses (pointers) are represented using only a subset of the 64 bits available in a machine word, which leave many bits unused. In this paper, we explore the use, not only of the least-significant bits of pointers, but also of the most-significant bits, for representing algebraic data types in a full ML compiler. It turns out that, with such a particular utilisation of otherwise unused bits, which we call double-ended bit-stealing, it is possible to choose unboxed representations for a large set of data types, while still not violating the principle of uniform data-representations. Examples include Patricia trees, union-find data structures, stream data types, internal language representations for types and expressions, and mutually recursive ASTs for full language definitions.

The double-ended bit-stealing technique is implemented in the MLKit compiler and speedup ranges from 0 to 26 percent on benchmarks that are influenced by the technique. For MLKit, which uses abstract data types extensively, compilation speedups of around 9 percent are achieved for compiling MLton (another Standard ML compiler) and for compiling MLKit itself.

CCS Concepts: • **Software and its engineering** → **Functional languages**; *Runtime environments.*

Additional Key Words and Phrases: Data-type representations, Unboxing, Compilation, Functional languages

ACM Reference Format:
Martin Elsman. 2024. Double-Ended Bit-Stealing for Algebraic Data Types. *Proc. ACM Program. Lang.* 8, ICFP, Article 239 (August 2024), 33 pages. https://doi.org/10.1145/3674628

1 Introduction

Functional programmers have long been accustomed to the possibility of using algebraic data types for modeling data and for designing and developing programs. Consider the following algebraic data type declarations:

```
datatype bop = ADD | MUL | SUB | DIV
datatype e = Var of string | Num of real | Tup of e list
           | Sel of int * e | Bin of bop * e * e
```

Author's Contact Information: Martin Elsman, University of Copenhagen, Copenhagen, Denmark, mael@di.ku.dk.

ACM 2475-1421/2024/8-ART239
https://doi.org/10.1145/3674628

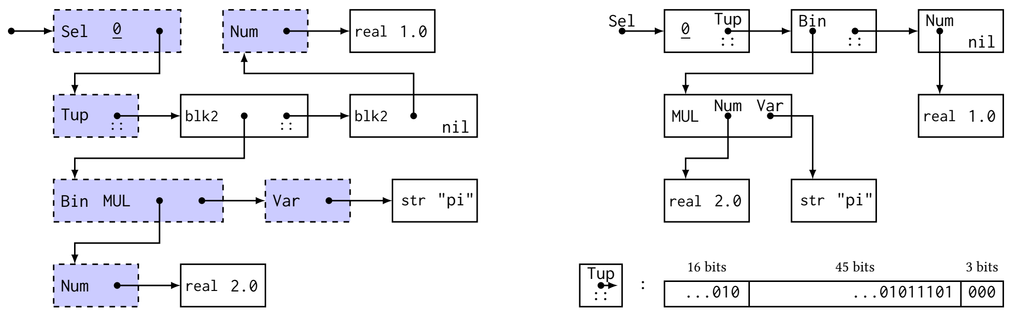

Fig. 1. Boxed and unboxed representations of a simple expression tree (the top two pointer graphs). The light-blue (and dashed) rectangular boxes represent boxed constructed values and the white rectangular boxes represent allocated pairs, boxed reals, and strings. An illustration of the word layout for the embedded Tup value is shown in the lower right part of the figure. Here the :: constructor occupies the 3 lower bits (000) and the Tup constructor occupies the 16 higher bits. The middle 45 bits represent the pointer to the pair carried by the :: constructor.

These declarations specify the types bop and e, an enumeration type for possible binary operations (e.g., ADD denotes addition) and an inductively defined type for expression values, including variables (i.e., Var), numbers (i.e., Num), tuples (i.e., Tup), tuple selections (i.e., Sel), and binary operations (i.e., Bin). Although the programmer may have an idea about how values of these types are represented at runtime, most ML compilers will make their own representation choices based on external functional needs such as simplicity, code reuse, efficient execution, low memory usage, support for reference-tracing garbage-collection, interoperability, and so on.

In strict ML-like languages, a common representation of the above algebraic data types is to represent the constructors for the bop type as unboxed integers and constructed values of the e type as pointers to blocks of consecutive words in memory, with the first word holding information that allows for discriminating between the different constructors, and the remaining words holding the *constructor payload*, which for the Bin constructor is three words, for the Sel constructor is two words, and for the other constructors is one word.[1] Consider the value

```
val ex : e = Sel (0, Tup [Bin (MUL, Num 2.0, Var "pi"), Num 1.0])
```

A graphical layout of the common boxed representation of the value is shown to the left in Figure 1. We see that for each of the boxed constructed values (shown dashed and in light-blue color), a tag word is preceding the constructor's payload. Lists are here represented unboxed, meaning that its constructors are represented unboxed, which relies on a common trick that makes use of so-called low bit-stealing to distinguish nil from cons-cells marked :: in the layout. A particular feature of an unboxed data type is that constructing a value of that type (possibly given an argument) does not require allocation. Notice that both strings and reals are represented boxed here, which is to make sure that a uniform representation is used for values, that is, all represented values are implemented uniformly and may be identified using one machine word.

This paper presents a new technique for representing algebraic data types, called *double-ended bit-stealing*, which uses not only the least-significant bits of pointers for distinguishing data constructors for simple data types such as lists (making traditional use of common pointer-alignment properties),

[1]Because the garbage collector needs to distinguish pointers from integers, integers are usually shifted by 1 with the least-significant bit set to one.

but also uses the most-significant bits of pointers. The technique is based on the property that the most-significant bits of pointers are freely available for use on most modern 64-bit architectures. For instance, on x86_64 architectures, the notion of canonical addresses [Intel 2024] ensures that the most significant 16 bits of a pointer are identical to bit 47, which is 0 for user addresses under Linux [Linux Kernel Development Community 2024]. Similar opportunities exist on other UNIX operating systems and on RISC-V and AArch64 platforms [Bradbury 2023].

As an example of double-ended bit-stealing, the value bound to the variable `ex` (defined earlier) may be represented as shown to the right in Figure 1. The notation used in the figure places so-called *high-bit* tags on top of pointer sources (e.g., the tag `Sel` is placed on top of the left-most arrow source in the layout). Similarly, so-called *low-bit* tags are placed below pointer sources (e.g., `nil` and `::`). An important difference between the two layouts is that the layout to the right treats all non-nullary constructors as constructors that take only one argument, which may be a tuple. We shall return to the consequences of this choice later. Another difference is that some of the tag words that also contain size-information about blocks of allocated memory do not appear in the layout to the right, which is a possible representation choice when, for small tuple values, a big-bag-of-pages (BIBOP) approach is used for memory management (allocation of small identically-sized blocks are grouped together and share a common tag, located, for instance, at a page-aligned boundary). In combination, the two techniques save almost half of the memory needed for representing the `ex` value. Notice how a `Tup`-value is represented using multiple levels of unboxing. The representation uses the high bits for specifying that it is a `Tup` value and the remainder of the bits to represent either `nil` or a possible `::` cell. Not only does double-ended bit-stealing lead to fewer memory accesses, the reduction in memory usage may also likely have a positive effect on cache performance.

The suggested representation technique has been implemented in the MLKit Standard ML compiler [Tofte et al. 2022], which, in general, uses a combination of region inference [Tofte and Talpin 1997] and reference-tracing garbage collection for memory management [Elsman and Hallenberg 2021; Hallenberg et al. 2002]. In this context, unboxing has an even bigger effect on the performance and memory usage of programs as regions for the otherwise boxed values cease to appear in the generated code. The consequence is that critical functions may take fewer regions as arguments and that the overall overhead of allocation decreases. The traditional unboxing strategy used by the MLKit (before this work) has been to use only the lower bits in pointers for tagging unboxed values, which limits what values can be represented unboxed, compared to the double-ended bit-stealing approach.

Background on Representing Algebraic Data Types. There are already many approaches to representing algebraic data types and we have already seen common representations for the `bop` and `e` data types. For representing nullary constructors, a common strategy is also to use an unboxed representation, where each nullary constructor is represented by a unique integer. This strategy, which is used for instance by OCaml and MosML, may sometimes be beneficial (i.e., it is quick to test for a particular nullary constructor) but also has the drawback that multiple tests are needed to test for a unary constructor. In particular, a pattern-matching implementation needs first to identify the constructor kind (boxed or unboxed) after which the implementation can determine the particular constructor value. Moreover, such a value representation occupies not only the pointer bits but also the least-significant bits of a machine word.

An alternative, which is the strategy used by the MLKit compiler, is to use a boxed representation also for nullary constructors (which may be allocated statically). The benefit of such a representation is that discriminating between match cases becomes simpler and that such fully boxed representations occupy only the pointer bits of a machine word, leaving both the lower and the higher bits to be used by constructors that take values of the boxed type as argument.

A common strategy is also to understand constructors that take a tuple or a record as argument as a constructor that takes multiple arguments and to represent such constructed values as the tuple with a prefix-tag that encodes the tag of the constructor and separately the size of the proceeding block. A common trick in this respect, shared by OCaml and MosML, is to place the block size information in the same way in tuples and in values constructed from multi-argument constructors. In this way, it is immediately possible, at runtime, to retrieve a pointer, representing a tuple, from a constructed value. Multi-argument constructors may introduce an additional cost, however. When the tuple is not constructed at the same time the constructed value is constructed, the tuple elements will need to be copied into the constructed value at the time the multi-argument constructed value is created.

Again, the MLKit compiler follows a different approach where constructors are either nullary or unary (also representation-wise), which may introduce unnecessary indirections, but which has the benefit that constructing a value from a tuple can be implemented without copying the tuple elements. In principle, there is no reason that both representation forms cannot coexist at runtime. In fact, OCaml (and MosML, which builds on the runtime system of the older CamlLight implementation [Leroy 1990]) supports the use of syntactic parentheses in data type bindings to specify that a constructor should be understood as a unary constructor and not a multi-argument constructor.

Reference tracing garbage collection usually constrains the representation of values as the collector needs to distinguish pointers to heap-allocated values (that must be traced) from other values (e.g., unboxed integers). Often, the least significant bit in values are used for this purpose, which, in a simple unboxing scheme allows types with two argument-taking constructors and a large number of nullary constructors to be represented unboxed. In practice, however, compilers for functional languages seldom goes beyond using more than one bit in pointers for tagging.

Historically, before 64 bits where used for pointers, no spare higher bits were available for tagging and when moving to 64-bit platforms, compiler implementors have been concerned about targeting 64-bit and 32-bit platforms in the same compiler code-base, which have put a constraint on the use of spare bits in 64-bit pointers. Moreover, when making use of spare bits in 64-bit pointers, implementations need to deal with complications in garbage collection algorithms, complications in foreign-function interfacing, and complications in code that exposes representation details to programmers. These issues can all be dealt with by proper abstraction mechanisms and, when needed, mechanisms for controlling representations of exposed data types [Eisenberg and Peyton Jones 2017].

Benefits of Double-Ended Bit-Stealing. There are several benefits to double-ended bit-stealing besides from directly saving space for constructor tagging. First, by using the high-end bits for storing constructor tags for an algebraic data type (or sum-type), we can use a type-based big-bag-of-pages (BIBOP) approach to represent constructor payloads compactly [Elsman and Hallenberg 2021]. Second, for ADTs with constructors that take only one argument (i.e., that do not take products as arguments), which are sometimes called single-argument constructors, we may (under certain conditions) represent constructed values unboxed (with no allocation). In such cases (examples include union-find data structures and streams), a double-ended bit-stealing representation is strictly more compact than a representation that encodes constructor tags in tuples, even without applying a big-bag-of-pages approach for avoiding tuple tags. We therefore conjecture that double-ended bit-stealing may be of general benefit for compiling and representing algebraic data types. Moreover, the technique is compatible also with boxing strategies that encode constructor tags for some data types in tuples.

Contributions. We claim the following contributions:

(1) We devise a technique for inferring uniform unboxity decisions for algebraic data types that utilise both the least-significant and the most-significant unused bits in pointers. The technique can be safely integrated with garbage collection.
(2) We present a formal static semantics and a formal dynamic semantics for a small language with algebraic data types and demonstrate type soundness of the language. Moreover, we present a formal statement saying that evaluating an expression under different unboxing strategies lead to similar results.
(3) We present an inference algorithm for double-ended bit-stealing, describe the details of a concrete implementation in a full Standard ML compiler, and discuss various unboxing trade-offs and how the programmer can potentially monitor and influence unboxing decisions by adhering to a set of simple unboxing rules.
(4) We report on a series of benchmarks measuring the effect of double-ended bit-stealing for representing algebraic data types in a full Standard ML compiler. In particular, for each benchmark, we report on the effect on execution time and memory usage and compare the results with execution times and memory usage for two other state-of-the-art Standard ML compilers (i.e., SML/NJ [Appel 1992] and MLton [Weeks 2006]) for the purpose of showing that the results obtained with MLKit are not off by orders of magnitudes compared with state-of-the-art compilers. Further, we report on the performance effects of double-ended bit-stealing on two larger programs, the MLKit itself and MLton. We also report on the double-ended bit-stealing decisions for the two large programs.

Outline. In the next section, we present a small polymorphic functional language equipped with algebraic data types (including pattern matching) and support for different representations of constructed data. We further present formal properties of the language. In Section 3, we present an inference algorithm for representing algebraic data types using double-ended bit-stealing, and argue that the inference leads to sound unboxing decisions. The inference algorithm finds a sound fix-point solution for a group of (possibly recursive) data type declarations based on boxity decisions for other data types the declarations refer to. In the worst case, all data types are represented boxed (which is always a sound solution). In Section 4, we present details of the implementation and in Section 5, we present a large series of real-world examples demonstrating double-ended bit-stealing. In Section 6, we present numbers demonstrating that the approach is beneficial in practice both with respect to memory usage and to its effect on execution time for a series of benchmarks, including the MLKit compiler and the MLton compiler. In Section 7, we describe related work and in Section 8, we conclude and discuss future work.

2 A Polymorphic Language with Algebraic Data Types

In this section, we present a polymorphic language with algebraic data types and a type system that allows for different boxity decisions for the algebraic data types, under some well-formedness constraints, which specify the boxity possibilities for the involved data types. We show type soundness for the language (Section 2.7) and demonstrate a simulation property (Section 2.8) stating that well-typed programs evaluate to the same result independent of which well-formed boxity decision is chosen. In the following sections, we first introduce the notions of types and type schemes, the notions of boxities and well-formed boxity environments, the notions of values, expressions, and programs, the type system for the language, and a dynamic semantics for the language. An overview of syntactic constructs and notation, introduced in this section, is available in Appendix A.3 (auxiliary material).

2.1 Types, Type Schemes, and Substitutions

We assume a denumerably infinite set of *type names*, each equipped with an *arity* (n), and ranged over by t. When we want to be explicit about the arity of a type name t, we write the type name on the form $t^{(n)}$. We also assume a denumerably infinite set of *type variables*, ranged over by α.

Types (τ) and type schemes (σ) are defined according to the following grammar:

$$\begin{array}{lcll} \tau & := & \alpha \mid \texttt{int} \mid \vec{\tau}\, t \mid \tau * \tau \mid \tau \rightarrow \tau & \text{types} \\ \sigma & := & \forall\vec{\alpha}.\tau & \text{type schemes} \end{array}$$

Whenever we have a type on the form $(\tau_1, \cdots, \tau_n)\ t$, it is implicitly assumed that the type name t has arity n. When $\forall\vec{\alpha}.\tau$ is a type scheme, we consider $\vec{\alpha}$ to be *bound* in τ. We consider type schemes identical up to renaming of bound variables. A *substitution* (S) is a finite map from type variables to types. The effect of applying a substitution S to an object X, written $S(X)$, is to replace (simultaneously) all free (i.e., non-bound) occurrences of type variables in X with corresponding types in S (extended to be the identity outside of its domain). When S and S' are substitutions and X is some object, we write $(S' \circ S)(X)$ to mean $S'(S(X))$. We also write ftv X to denote the free type variables in X. When σ is some type scheme $\forall\vec{\alpha}.\tau$, we say that a type τ' is an *instance* of σ, written $\sigma \geq \tau'$, if there exists a substitution S such that Dom $S = \{\vec{\alpha}\}$ and $S(\tau) = \tau'$.

2.2 Boxities and Well-Formed Boxity Environments

We define the notion of *boxity* (κ) according to the following grammar:

$$\kappa \quad ::= \quad \texttt{hub} \mid \texttt{lub} \mid \texttt{box} \qquad \text{boxities}$$

Intuitively, the box boxity specifies that a value is represented boxed, the lub boxity specifies that a value is represented unboxed using lower bits for tagging (low-unboxed), and the hub boxity specifies that a value is represented unboxed using the higher bits for tagging (high-unboxed). Boxities form a total order ($>$) specified by hub $>$ lub $>$ box, indicating that the hub boxity uses strictly more bits than the lub boxity, which uses strictly more bits than the box boxity.

We assume a denumerably infinite number of *value constructors*, ranged over by c. A *constructor environment* (C) is a finite map from value constructors to type schemes and a *boxity environment* (B) is a finite map from type names to pairs of a boxity and a constructor environment. In general, when M and M' are finite maps with Dom $M \cap$ Dom $M' = \emptyset$, we write M, M' to denote the finite map with domain Dom $M \cup$ Dom M' and values $(M, M')(x) = M(x)$, if $x \in$ Dom M, and $(M, M')(x) = M'(x)$, if $x \in$ Dom M'. Moreover, we write fc B to denote the free constructors in B, that is the set $\{c | c \in \text{Dom}\ C, (\kappa, C) = B(t), t \in \text{Dom}\ B\}$.

We shall now prepare for defining what constitutes a well-formed boxity environment. We first introduce a relation that establishes a conservative approximation to the boxity of a type. The relation is defined as a set of inference rules allowing inferences of judgments of the form $B \vdash \tau : \kappa$, which are read "the type τ has boxity κ in the boxity environment B":

Type boxity $\boxed{B \vdash \tau : \kappa}$

$$\frac{}{B \vdash \alpha : \texttt{hub}} \qquad \frac{}{B \vdash \texttt{int} : \texttt{hub}} \qquad \frac{}{B \vdash \tau_1 * \tau_2 : \texttt{box}} \qquad \frac{}{B \vdash \tau \rightarrow \tau' : \texttt{box}}$$

$$\frac{B(t) = (\kappa, C)}{B \vdash \vec{\tau}\, t : \kappa} \qquad \frac{B \vdash \tau : \kappa \quad \kappa' > \kappa}{B \vdash \tau : \kappa'}$$

In the case of constructed types of the form $\vec{\tau}\ t$, the relation establishes the boxity of the type by looking up the type name t in B. Notice that type variables have boxity hub, which is to make sure that no matter what type is instantiated for the type variable, no bits are used in the word-representation of values of that type to tag constructed values. The last rule says that it is always safe to assume that values of a type use more bits than it actually does.

We now define a relation specifying that, given a boxity environment B, the boxity specification of the type name t is *well-formed* in B, written $B \vdash t$. The relation is defined by the following rules, where $H = 16$ represents the number of bits available for high-bit tagging:[2]

Well-Formed Type Name Bindings $\boxed{B \vdash t}$

$$\frac{\begin{array}{c} C = \{c_1 : \forall\vec{\alpha}.\tau_1 \to \vec{\tau}\,t, \ldots, c_m : \forall\vec{\alpha}.\tau_m \to \vec{\tau}\,t, c_{m+1} : \sigma_0, \ldots, c_{m+n} : \sigma_0\} \\ B(t) = (\mathsf{hub}, C) \quad B \vdash \tau_1 : \mathtt{lub} \quad \ldots \quad B \vdash \tau_m : \mathtt{lub} \quad m + n < 2^H \quad \sigma_0 = \forall\vec{\alpha}.\vec{\tau}\,t \end{array}}{B \vdash t}\;[\textsc{hub}]$$

$$\frac{\begin{array}{c} C = \{c_1 : \forall\vec{\alpha}.\tau \to \vec{\tau}\,t, c_2 : \sigma_0, \ldots, c_n : \sigma_0\} \\ B(t) = (\mathtt{lub}, C) \quad B \vdash \tau : \mathsf{box} \quad \sigma_0 = \forall\vec{\alpha}.\vec{\tau}\,t \end{array}}{B \vdash t}\;[\textsc{lub}] \qquad \frac{\begin{array}{c} C = \{c_1 : \sigma_0, \ldots,\ c_n : \sigma_0\} \\ B(t) = (\mathtt{lub}, C) \quad \sigma_0 = \forall\vec{\alpha}.\vec{\tau}\,t \end{array}}{B \vdash t}\;[\textsc{enum}]$$

$$\frac{\begin{array}{c} C = \{c_1 : \forall\vec{\alpha}.\tau_1 \to \vec{\tau}\,t, \ldots, c_m : \forall\vec{\alpha}.\tau_m \to \vec{\tau}\,t, c_{m+1} : \sigma_0, \ldots, c_{m+n} : \sigma_0\} \\ B(t) = (\mathsf{box}, C) \quad B \vdash \tau_1 : \kappa_1 \quad \ldots \quad B \vdash \tau_m : \kappa_m \quad \sigma_0 = \forall\vec{\alpha}.\vec{\tau}\,t \end{array}}{B \vdash t}\;[\textsc{box}]$$

We further say that a boxity environment is *well-formed*, written $\vdash B$, if for all $t \in \mathrm{Dom}\,B$, we have $B \vdash t$ and we also have $B(t) = (\kappa, C)$ implies $\mathrm{Dom}\,C \cap \mathrm{fc}(B \setminus \{t\}) = \emptyset$ (here we write $B \setminus \{t\}$ to denote the boxity environment B restricted to the domain $\mathrm{Dom}\,B \setminus \{t\}$).

The notion of well-formed boxity environments captures the necessary conditions for constructed values to be distinguished effectively at runtime. For simplicity and without lack of generality, we assume that each constructor can belong to at most one data type. It is straightforward to verify that boxity environments that map all type names to the boxity box are well-formed (rule [box]). On the other hand, it is also straightforward to come up with boxity environments that are not well-formed. One example is a boxity environment that maps the `option` type name to the `lub` boxity, which is problematic because the `SOME` constructor has the type scheme $\forall\alpha.\alpha \to \alpha$ `option` and [lub] requires the argument type to the unary constructor to be box, as the least significant bits must be reserved to discriminate between the two constructors.[3] Unfortunately, because type variable can be substituted with any type, type variables have type boxity hub, which does not align with the assumption in [lub].

2.3 Data-Type Bindings and Data-Type Declarations

We define the notion of *data-type bindings* (*db*) and the notion of *data-type declarations* (*dd*) as follows:

db	::=	$\vec{\alpha}\ t = C$	data-type bindings
dd	::=	`type` db_1 `and` ... `and` db_n	data-type declarations

Notice that we assume here that the syntactic constructor specifications have been elaborated into constructor environments mapping constructors into type scheme. Further, notice that data-type declarations allow for multiple types to be declared simultaneously, which allows for mutually recursive bindings.

Data-type bindings and data-type declarations are *elaborated* (i.e., checked) by two rules that allow inferences of judgments of the forms $B \vdash db : B'$ and $B \vdash dd : B'$, which are read "the data-type binding db (or data-type declaration dd) elaborates to the boxity environment B' in B".

[2]A lower value of H may be used for platforms that use more than 48 bits for representing pointers.
[3]Besides the `SOME` constructor, the `option` type also has the nullary constructor `NONE`, which has type scheme $\forall\alpha.\alpha\ t$.

Elaboration of data-type bindings $\boxed{B \vdash db : B'}$

$$\frac{B(t) = (\kappa, C) \quad B \vdash t \quad \text{arity}(t) = n}{B \vdash \vec{\alpha}^{(n)}\ t = C : \{t \mapsto (\kappa, C)\}}\ [\textsc{e-db}]$$

Elaboration of data-type declarations $\boxed{B \vdash dd : B'}$

$$\frac{\begin{array}{c}B, B' \vdash db_1 : B_1 \quad \ldots \quad B, B' \vdash db_n : B_n \\ B' = B_1, \ldots, B_n \qquad \text{fc } B_i \cap (\text{fc } B_j \cup \text{fc } B) = \emptyset,\ \text{for all } i, j \text{ s.t. } i \neq j\end{array}}{B \vdash \texttt{type}\ db_1\ \texttt{and}\ \ldots\ \texttt{and}\ db_n : B'}\ [\textsc{e-dd}]$$

The rule for data-type bindings simply requires the type name t to be well-formed in B and occurring in B with the given constructor environment. The rule for data-type declarations enforces uniqueness of constructor names and declared type names. Notice that for the rule for data-type declarations, given B and dd, there may be multiple different B' so that $B \vdash dd : B'$ holds.

The following proposition states that data-type declarations elaborate to well-formed boxity environments:

Proposition 2.1 (Well-formed data-type declarations yields well-formed boxity-environments). *If* $\vdash B$ *and* $B \vdash dd : B'$ *then* $\vdash B, B'$.

The proof follows from the definition of elaboration of data-type declarations, which ensures that constructors are unique for each data-type binding and that $B, B' \vdash t_i$ holds for each t_i introduced by the involved data-type bindings. It follows that $\vdash B, B'$ holds as required.

2.4 Values, Expressions, and Programs

Values are divided into *boxed values* (l), *low unboxed values* (v), and *high unboxed values* (w):

$$\begin{array}{lcll} l & ::= & \langle w, w\rangle \mid \langle \lambda x.e\rangle \mid \langle \texttt{fix}\ f\ x = e\rangle \mid \{c, w\} \mid \{c\} & \text{boxed values} \\ v & ::= & l \mid l{\cdot}c \mid {\cdot}c & \text{low-unboxed values} \\ w & ::= & v \mid c{\cdot}v \mid c{\cdot} \mid d & \text{high-unboxed values} \end{array}$$

We use the letter l to denote boxed values as to indicate that boxed values are represented by locations (pointers to objects) in the heap. Boxed values include pairs $\langle w, w\rangle$, closures $\langle \lambda x.e\rangle$, recursive closures $\langle \texttt{fix}\ f\ x = e\rangle$, boxed unary-constructed values $\{c, w\}$, and boxed nullary-constructed values $\{c\}$.

Low unboxed values v include those values that occupy the same number of bits as boxed values (i.e., pointers). Besides from boxed values, low unboxed values also include low-tagged pointers $l{\cdot}c$, where c is a constructor stored in the least significant bits of a pointer and low-unboxed nullary constructors ${\cdot}c$, which are represented with the least-significant bit set (to distinguish them from unary constructors) and may use other bits occupied by low-unboxed values.

High-unboxed values w include those values that may occupy all bits in a word. Such values include low-unboxed values, integers d (using perhaps all 64 bits), high-tagged low-unboxed values $c{\cdot}v$, where c is a constructor stored in the most-significant bits of a word, and high-unboxed nullary constructors $c{\cdot}$, which are represented using the most-significant bits of a word.

Notice, in particular, that the high-unboxed value $c_{\texttt{T}}{\cdot}(\langle 23, {\cdot}c_{\texttt{nil}}\rangle{\cdot}c_{\texttt{::}})$ may represent the constructed value `T[23]` if $c_{\texttt{T}}$ represents the source-level constructor `T` and if $c_{\texttt{nil}}$ and $c_{\texttt{::}}$ represent the source-level constructors `nil` and `::`, respectively, given an appropriate boxity environment.

Expressions e, patterns p, and programs P are defined as follows:

$$\begin{array}{llll}
e & ::= & x \mid w \mid c\ e \mid \underline{c} \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid (e, e) \mid \#i\ e & \text{expressions} \\
 & \mid & \lambda x.e \mid \texttt{fix}\ f\ x = e \mid \texttt{case}\ e\ \texttt{of}\ b \mid \ldots \mid b & \\
b & ::= & p \Rightarrow e & \text{branches} \\
p & ::= & c\ x \mid c & \text{patterns} \\
P & ::= & dd\ P \mid e & \text{programs}
\end{array}$$

Expressions of the form $\underline{c}$ denote a use of a nullary constructor c where expressions of the form $c\ e$ denote a use of a unary constructor (applied to its argument). These aspects are enforced by the typing rules, presented in the next section. Expressions of the form $\#i\ e$ denote projections from pairs.

For expressions of the form $\lambda x.e$ (and $\texttt{fix}\ f\ x = e$) and for values of the form $\langle \lambda x.e \rangle$ (and $\langle \texttt{fix}\ f\ x = e \rangle$), we consider x (and f) *bound* in e. For expressions of the form $\texttt{let}\ x = e\ \texttt{in}\ e'$, we consider x *bound* in e'. For branches of the form $c\ x \Rightarrow e$, we consider x *bound* in e. All values and expressions are considered equal up to renaming of bound variables.

As a syntactic restriction, we enforce that for expressions of the form $\texttt{case}\ e\ \texttt{of}\ b_1 \mid \ldots \mid b_n$, the branches $b_1, \ldots, b_n$ branch on distinct constructors. In other words, there are no redundant matches. We shall later see that the typing rules will enforce that `case` constructs are exhaustive.

2.5 Well-Typed Values, Expressions, and Programs

Environments (Γ) are finite maps from program variables to type schemes. When B is some boxity environment, p is some pattern, and $\vec{\tau}\ t$ is some constructed type, we write $(B, \vec{\tau}\ t) \triangleright p$ to denote the empty environment if p is a nullary constructor and the environment $\{x : \tau\}$ if p is on the form $c\ x$ and $B(t)(c) \geq \tau \rightarrow \vec{\tau}\ t$.

The typing rules, which are shown in Figure 2, are divided into rules for typing values, rules for typing expressions, and rules for typing programs. Whereas the typing rules for values take the form $B \vdash w : \sigma$, the typing rules for expressions take the form $B, \Gamma \vdash e : \sigma$, which are abbreviated to $B \vdash e : \sigma$ if Γ is the empty map.

Most of the rules are standard, in particular the rules for expressions. Without lack of generality, the language supports generalisation only through `fix` expressions and through `fix` values; other polymorphic bindings of functions (i.e., lambda-functions) can be turned into bindings of non-recursive `fix` constructs.

The typing rules for constructed values link the concrete value representation with the boxity for the constructor specified in the boxity environment. The typing rules for programs ensure that the program expression is type checked in boxity environments resulting from the elaborated data-type declarations of a program, which ensures that program expressions are type checked in well-formed boxity environments. For brevity, we have omitted straightforward fall-through rules from Figure 2. For instance, we implicitly have that if $B, \Gamma \vdash v : \sigma$ then $B, \Gamma \vdash w : \sigma$ if $w = v$.

It is an essential property that the typing rules are closed under substitution, which is stated by the following proposition:

Proposition 2.2 (Typing Rules Closed Under Type Substitution). *If $B, \Gamma \vdash e : \sigma$ then $B, S(\Gamma) \vdash e : S(\sigma)$, for any substitution S.*

The proposition relies on a series of properties, including the facts that boxity environments are always closed with respect to type variables, that type scheme instantiation is closed under substitution, and that, for any substitution S, we have that $\Gamma = (B, \vec{\tau}\ t) \triangleright p$ implies $S(\Gamma) = S(B, \vec{\tau}\ t) \triangleright p$, for any environment Γ, boxity environment B, type $\vec{\tau}\ t$, and pattern p.

Boxed values $\boxed{B \vdash l : \sigma}$

$$\frac{B \vdash w_1 : \tau_1 \quad B \vdash w_2 : \tau_2}{B \vdash \langle w_1, w_2 \rangle : \tau_1 * \tau_2}\ [\textsc{l-pair}] \qquad \frac{B, \{x : \tau\} \vdash e : \tau'}{B \vdash \langle \lambda x.e \rangle : \tau \to \tau'}\ [\textsc{l-lam}]$$

$$\frac{B, \{x : \tau, f : \tau \to \tau'\} \vdash e : \tau'}{B \vdash \langle \texttt{fix}\ f\ x = e \rangle : \forall \vec{\alpha}.\tau \to \tau'}\ [\textsc{l-fix}]$$

$$\frac{\begin{array}{c} B(t) = (\mathsf{box}, C) \\ C(c) \geq \tau \to \vec{\tau}\ t \quad B \vdash w : \tau \end{array}}{B \vdash \{c, w\} : \vec{\tau}\ t}\ [\textsc{l-c1}] \qquad \frac{B(t) = (\mathsf{box}, C) \quad C(c) \geq \vec{\tau}\ t}{B \vdash \{c\} : \vec{\tau}\ t}\ [\textsc{l-c0}]$$

Low-unboxed values $\boxed{B \vdash v : \sigma}$

$$\frac{\begin{array}{c} B(t) = (\mathsf{lub}, C) \\ C(c) \geq \tau \to \vec{\tau}\ t \quad B \vdash l : \tau \end{array}}{B \vdash l{\cdot}c : \vec{\tau}\ t}\ [\textsc{v-c1}] \qquad \frac{B(t) = (\mathsf{lub}, C) \quad C(c) \geq \vec{\tau}\ t}{B \vdash {\cdot}c : \vec{\tau}\ t}\ [\textsc{v-c0}]$$

High-unboxed values $\boxed{B \vdash w : \sigma}$

$$\frac{\begin{array}{c} B(t) = (\mathsf{hub}, C) \\ C(c) \geq \tau \to \vec{\tau}\ t \quad B \vdash v : \tau \end{array}}{B \vdash c{\cdot}v : \vec{\tau}\ t}\ [\textsc{w-c1}] \qquad \frac{\begin{array}{c} B(t) = (\mathsf{hub}, C) \\ C(c) \geq \vec{\tau}\ t \end{array}}{B \vdash c{\cdot} : \vec{\tau}\ t}\ [\textsc{w-c0}] \qquad \frac{}{B \vdash d : \texttt{int}}\ [\textsc{w-int}]$$

Expressions $\boxed{B, \Gamma \vdash e : \sigma}$

$$\frac{B, \Gamma, x : \tau \vdash e : \tau'}{B, \Gamma \vdash \lambda x.e : \tau \to \tau'}\ [\textsc{e-lam}] \qquad \frac{\Gamma(x) = \sigma}{B, \Gamma \vdash x : \sigma}\ [\textsc{e-var}] \qquad \frac{B \vdash w : \sigma}{B, \Gamma \vdash w : \sigma}\ [\textsc{e-val}]$$

$$\frac{B, \Gamma \vdash e : \sigma \quad \sigma \geq \tau}{B, \Gamma \vdash e : \tau}\ [\textsc{e-inst}] \qquad \frac{B, \Gamma \vdash e_1 : \tau_1 \quad B, \Gamma \vdash e_2 : \tau_2}{B, \Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}\ [\textsc{e-pair}]$$

$$\frac{B(t)(c) \geq \tau \to \vec{\tau}\ t \quad B, \Gamma \vdash e : \tau}{B, \Gamma \vdash c\ e : \vec{\tau}\ t}\ [\textsc{e-con1}] \qquad \frac{B(t)(c) \geq \vec{\tau}\ t}{B, \Gamma \vdash \underline{c} : \vec{\tau}\ t}\ [\textsc{e-con0}]$$

$$\frac{B, \Gamma \vdash e : \tau_1 * \tau_2 \quad i \in \{1, 2\}}{B, \Gamma \vdash \#i\ e : \tau_i}\ [\textsc{e-sel}] \qquad \frac{B, \Gamma \vdash e_1 : \sigma \quad B, \Gamma, x : \sigma \vdash e_2 : \sigma'}{B, \Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \sigma'}\ [\textsc{e-let}]$$

$$\frac{\begin{array}{c} B, \Gamma \vdash e : \tau \to \tau' \\ B, \Gamma \vdash e' : \tau \end{array}}{B, \Gamma \vdash e\ e' : \tau'}\ [\textsc{e-app}] \qquad \frac{B, \Gamma, f : \tau \to \tau', x : \tau \vdash e : \tau' \quad \{\vec{\alpha}\} \cap \mathrm{ftv}\,\Gamma = \emptyset}{B, \Gamma \vdash \texttt{fix}\ f\ x = e : \forall \vec{\alpha}.\tau \to \tau'}\ [\textsc{e-fix}]$$

$$\frac{\begin{array}{c} \Gamma_1 = (B, \vec{\tau}\ t) \triangleright p_1 \quad \ldots \quad \Gamma_n = (B, \vec{\tau}\ t) \triangleright p_n \quad B(t) = (\kappa, C) \quad |\mathrm{Dom}\ C| = n \\ B, \Gamma \vdash e : \vec{\tau}\ t \quad B, \Gamma, \Gamma_1 \vdash e_1 : \tau \quad \ldots \quad B, \Gamma, \Gamma_n \vdash e_n : \tau \end{array}}{B, \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n : \tau}\ [\textsc{e-case}]$$

Programs $\boxed{B \vdash P : \tau}$

$$\frac{B \vdash dd : B' \quad B, B' \vdash P : \tau}{B \vdash dd\ P : \tau}\ [\textsc{p-dat}] \qquad \frac{B, \{\} \vdash e : \tau}{B \vdash e : \tau}\ [\textsc{p-exp}]$$

Fig. 2. Typing rules for values, expressions, and programs.

Evaluation of expressions $\boxed{B \vdash e \rightsquigarrow e'}$

$$\frac{B \vdash e \rightsquigarrow e' \quad E \neq [\cdot]}{B \vdash E[e] \rightsquigarrow E[e']}\ [\textsc{r-ctx}] \qquad \frac{i \in \{1,2\}}{B \vdash \#i\ \langle w_1, w_2 \rangle \rightsquigarrow w_i}\ [\textsc{r-sel}]$$

$$\frac{}{B \vdash \lambda x.e \rightsquigarrow \langle \lambda x.e \rangle}\ [\textsc{r-lam}] \qquad \frac{}{B \vdash \texttt{fix}\ f\ x = e \rightsquigarrow \langle \texttt{fix}\ f\ x = e \rangle}\ [\textsc{r-fix}]$$

$$\frac{}{B \vdash (w, w') \rightsquigarrow \langle w, w' \rangle}\ [\textsc{r-pair}] \qquad \frac{}{B \vdash \langle \texttt{fix}\ f\ x = e \rangle\ w \rightsquigarrow e[(\texttt{fix}\ f\ x = e)/f, w/x]}\ [\textsc{r-appf}]$$

$$\frac{}{B \vdash \langle \lambda x.e \rangle\ w \rightsquigarrow e[w/x]}\ [\textsc{r-app}] \qquad \frac{}{B \vdash \texttt{let}\ x = w\ \texttt{in}\ e \rightsquigarrow e[w/x]}\ [\textsc{r-let}]$$

$$\frac{\mathit{ckind}(B,c) = \mathsf{box} \quad b_i = c\ x \Rightarrow e \quad 1 \le i \le n}{B \vdash \mathsf{case}\ \{c, w\}\ \mathsf{of}\ b_1 \mid \ldots \mid b_n \rightsquigarrow e[w/x]}\ [\textsc{r-bcase1}] \qquad \frac{\mathit{ckind}(B,c) = \mathsf{box}}{B \vdash c\ w \rightsquigarrow \{c, w\}}\ [\textsc{r-bc1}]$$

$$\frac{\mathit{ckind}(B,c) = \mathsf{box} \quad b_i = c \Rightarrow e \quad 1 \le i \le n}{B \vdash \mathsf{case}\ \{c\}\ \mathsf{of}\ b_1 \mid \ldots \mid b_n \rightsquigarrow e}\ [\textsc{r-bcase0}] \qquad \frac{\mathit{ckind}(B,c) = \mathsf{box}}{B \vdash \underline{c} \rightsquigarrow \{c\}}\ [\textsc{r-bc0}]$$

$$\frac{\mathit{ckind}(B,c) = \texttt{lub} \quad b_i = c\ x \Rightarrow e \quad 1 \le i \le n}{B \vdash \mathsf{case}\ l{\cdot}c\ \mathsf{of}\ b_1 \mid \ldots \mid b_n \rightsquigarrow e[l/x]}\ [\textsc{r-lcase1}] \qquad \frac{\mathit{ckind}(B,c) = \texttt{lub}}{B \vdash c\ l \rightsquigarrow l{\cdot}c}\ [\textsc{r-lc1}]$$

$$\frac{\mathit{ckind}(B,c) = \texttt{lub} \quad b_i = c \Rightarrow e \quad 1 \le i \le n}{B \vdash \mathsf{case}\ {\cdot}c\ \mathsf{of}\ b_1 \mid \ldots \mid b_n \rightsquigarrow e}\ [\textsc{r-lcase0}] \qquad \frac{\mathit{ckind}(B,c) = \texttt{lub}}{B \vdash \underline{c} \rightsquigarrow {\cdot}c}\ [\textsc{r-lc0}]$$

$$\frac{\mathit{ckind}(B,c) = \mathsf{hub} \quad b_i = c\ x \Rightarrow e \quad 1 \le i \le n}{B \vdash \mathsf{case}\ c{\cdot}v\ \mathsf{of}\ b_1 \mid \ldots \mid b_n \rightsquigarrow e[v/x]}\ [\textsc{r-hcase1}] \qquad \frac{\mathit{ckind}(B,c) = \mathsf{hub}}{B \vdash c\ v \rightsquigarrow c{\cdot}v}\ [\textsc{r-hc1}]$$

$$\frac{\mathit{ckind}(B,c) = \mathsf{hub} \quad b_i = c \Rightarrow e \quad 1 \le i \le n}{B \vdash \mathsf{case}\ c{\cdot}\ \mathsf{of}\ b_1 \mid \ldots \mid b_n \rightsquigarrow e}\ [\textsc{r-hcase0}] \qquad \frac{\mathit{ckind}(B,c) = \mathsf{hub}}{B \vdash \underline{c} \rightsquigarrow c{\cdot}}\ [\textsc{r-hc0}]$$

Fig. 3. Small-step reduction rules.

2.6 Evaluation

Before we can define reduction rules for the language, we first define the grammar for *redexes* (r) and *evaluation contexts* (E), which are given as follows:

$$\begin{array}{rcl} r & ::= & \mathsf{case}\ w\ \mathsf{of}\ b_1 \mid \ldots \mid b_n \\ & \mid & c\ w \mid \underline{c} \mid \#i\ w \\ & \mid & \texttt{let}\ x = w\ \texttt{in}\ e \\ & \mid & \lambda x.e \mid \texttt{fix}\ f\ x = e \\ & \mid & \langle \lambda x.e \rangle\ w \mid \langle \texttt{fix}\ f\ x = e \rangle\ w \end{array} \qquad \begin{array}{rcl} E & ::= & [.] \\ & \mid & \mathsf{case}\ E\ \mathsf{of}\ b_1 \mid \ldots \mid b_n \\ & \mid & \texttt{let}\ x = E\ \texttt{in}\ e \\ & \mid & (E, e) \mid (w, E) \mid c\ E \\ & \mid & E\ e \mid w\ E \mid \#i\ E \end{array}$$

When E is an evaluation context and e is an expression, we write $E[e]$ to denote the expression formed by filling the hole $[.]$ in the context E with the expression e. A *redex* (r) is an expression of a form that immediately matches the left-hand side of a reduction rule. A *value substitution* $[w_1/x_1, \ldots, w_n/x_n]$ maps program variables to high-unboxed values. We write $\mathit{ckind}(B, c) = \kappa$ to mean there exists t such that $B(t) = (\kappa, C)$ and $c \in \mathrm{Dom}\ C$. Reduction is defined in Figure 3 according to rules of the form $B \vdash e \rightsquigarrow e'$.

The first four lines of rules are standard and specify contextual reduction, tuple-selection, value-reduction, beta-reduction, and `let`-reduction. The following two lines of rules specify how constructor expressions reduces to values under particular boxity environments. The last three lines of rules specify pattern matching under particular boxity environments. For the two `box` cases, we make use of the assumption that it is possible to distinguish values of the form $\{c, w\}$ from values of the form $\{c\}$, which is possible by an implementation (using only a few machine instructions) because both boxed values created with nullary constructors and boxed values created with unary constructors are implemented as a pointer to a word containing the constructor tag (and because we know statically which constructors are nullary). For the two `lub` cases, we make use of the property that it is possible to distinguish values on the form $l{\cdot}c$ and ${\cdot}c$, which is possible by an implementation (using only a few machine instructions) because values of the form $l{\cdot}c$ do not make use of the least-significant bit. Finally, for the two `hub` cases, we make use of the property that it is possible to distinguish values of the form $c{\cdot}v$ from values of the form $c{\cdot}$, which is possible by an implementation (using only a few machine instructions) simply by distinguishing the constructors, which we know belong to the same data type.

We also remark here that in a compiled setting, we refine the `case` construct further and implement the different cases differently based on the boxity of the type of the expression that the `case` construct treats.

2.7 Soundness

The type safety property that we shall state is proven using well-known techniques for proving type safety [Morrisett 1995; Wright and Felleisen 1994]. We highlight the main properties below. Details of the proofs are provided in Appendix A.1 (auxiliary material). The proofs are mostly straightforward.

The first property states that a well-typed expression is either a value or can be separated into an evaluation context and a redex, which is demonstrated by induction over the typing derivation:

Proposition 2.3 (Unique Decomposition). *If $B \vdash e : \sigma$ then either e is a value or there exists a unique redex r, a type scheme σ', and a context E such that $B \vdash r : \sigma'$ and $e = E[r]$.*

Natural properties about contextual typing and value substitution hold, which are demonstrated by induction over the structure of E and e, respectively:

Proposition 2.4 (Context). *If $B \vdash E[e'] : \sigma$ then $B \vdash e' : \sigma'$ for some σ'. Further, if $B \vdash E[e] : \sigma$ and $B \vdash e : \sigma'$ and $B \vdash e' : \sigma'$ then $B \vdash E[e'] : \sigma$.*

Proposition 2.5 (Value Substitution). *If $B, \Gamma, x : \sigma' \vdash e : \sigma$ and $B \vdash w : \sigma'$ then $B, \Gamma \vdash e[w/x] : \sigma$.*

The latter proposition makes essential use of an "environment extension property" and of Barendregt's convention for renaming bound variables in expressions to avoid variable capture.

Type preservation (i.e., subject reduction) and progress are stated as follows and shown by induction on the structure of the typing derivation of e and by use of Proposition 2.3, Proposition 2.4, and Proposition 2.5:

Proposition 2.6 (Type Preservation). *If $\vdash B$ and $B \vdash e : \tau$ and $B \vdash e \rightsquigarrow e'$ then $B \vdash e' : \tau$.*

Proposition 2.7 (Progress). *If $\vdash B$ and $B \vdash e : \tau$ then either e is a value or there exists e' such that $B \vdash e \rightsquigarrow e'$.*

2.8 Simulation

In this section, we demonstrate a simulation property that states that program evaluations that differ only in their choice of value representations result in values that differ only with respect to boxity choices.

We first define a notion of similarity that relates values and expressions under different boxity environments. The similarity relation is defined in Figure 4. It is defined by a set of inductive judgments of the form $(B; e) \sim (B'; e')$, where B and B' are boxity environments and e and e' are expressions, high-unboxed values, low-unboxed values, or boxed values.

Many of the rules (e.g., rule [s-let], [s-tup], [s-lam], [s-fix], and [s-case]) are structural rules expressing that expressions are similar if their subexpressions are similar (under the different boxity environments). Rule [s-sym] expresses directly that the similarity relation is symmetric, which allows us to specify fewer rules than otherwise necessary. The interesting rules are those that relate constructed values, which lookup constructor representations in the respective boxity environments. There are six rules relating values constructed from nullary constructors ([s-h0], [s-l0], [s-b0], [s-hb0], [s-hl0], and [s-bl0]) and six rules relating values constructed from unary constructors ([s-h1], [s-l1], [s-b1], [s-hb1], [s-hl1], and [s-bl1]).

The following proposition states that simulation is closed under value substitution, which follows by a simple inductive argument with the only interesting case being the case for variables (i.e., rule [s-var]). Details of the proofs for the statements presented in this section are provided in Appendix A.2 (auxiliary material).

Proposition 2.8 (Simulation Value Substitution). *If* $(B; e) \sim (B'; e')$ *and* $(B; w) \sim (B'; w')$ *then* $(B; e[w/x]) \sim (B'; e'[w'/x])$.

We further introduce a notion of similarity between boxity environments. We say that a boxity environment B' *simulates* another boxity environment B, written $B \sim B'$, if $\vdash B$ and $\vdash B'$ and $\mathrm{Dom}\, B = \mathrm{Dom}\, B'$, and, for all $t \in \mathrm{Dom}\, B$, we have $(\kappa, C) = B(t)$ and $(\kappa', C') = B'(t)$ and $C = C'$, for some C, C', κ, and κ'. In other words, similarity of boxity environments allows for type names to differ only in their choice of well-formed boxity decisions. Simulation of boxity environments is symmetric.

We now introduce a notion of strong similarity, which, as simulation, is a symmetric relation relating pairs of boxity environments and expressions. A pair of a boxity environment and an expression $(B; e)$ is *strongly similar* to another pair of a boxity environment and an expression $(B'; e')$, written $(B; e) \approx (B'; e')$, if $B \sim B'$ and $(B; e) \sim (B'; e')$.

We can now state the following type-preservation property:

Proposition 2.9 (Typing Preserved by Strong Simulation). *If* $B, \Gamma \vdash e : \sigma$ *and* $(B; e) \approx (B'; e')$ *then* $B', \Gamma \vdash e' : \sigma$.

The proof of the above proposition is based on induction on the typing derivation for e and relies in essential ways on the definition of the similarity relation on values and on the similarity relation on boxity environments. Each of the three representations of argument-carrying constructors results in three subcases (utilising the corresponding simulation rule and, perhaps, the symmetry rule [s-sym] for simulation), which are shown, straightforwardly, each using a single induction step. The only other interesting case is the case for variables, which is easily shown by the fact that simulation is reflexive in the variable case (rule [s-var]).

The following proposition states that "similar well-typed expressions evaluate to similar results, in similar boxing environments":

Proposition 2.10 (Evaluation Preserved by Strong Simulation). *If* $B \vdash e : \tau$ *and* $(B; e) \approx (B'; e')$ *and* $B \vdash e \rightsquigarrow e_2$ *then there exists* e_2' *st* $B' \vdash e' \rightsquigarrow e_2'$ *and* $(B; e_2) \approx (B'; e_2')$.

Values $\boxed{(B; w) \sim (B'; w')}$

$$\frac{(B; w_1) \sim (B'; w'_1) \quad (B; w_2) \sim (B'; w'_2)}{(B; \langle w_1, w_2 \rangle) \sim (B'; \langle w'_1, w'_2 \rangle)}\;[\textsc{s-vtup}] \qquad \frac{(B; e) \sim (B'; e')}{(B; \langle \lambda x.e \rangle) \sim (B'; \langle \lambda x.e' \rangle)}\;[\textsc{s-vlam}]$$

$$\frac{(B; e) \sim (B'; e')}{(B; \langle \texttt{fix}\ f\ x = e \rangle) \sim (B'; \langle \texttt{fix}\ f\ x = e' \rangle)}\;[\textsc{s-vfix}] \qquad \frac{}{(B; d) \sim (B'; d)}\;[\textsc{s-int}]$$

$$\frac{ckind(B, c) = ckind(B', c) = \mathsf{hub}}{(B; c\cdot) \sim (B'; c\cdot)}\;[\textsc{s-h0}] \qquad \frac{ckind(B, c) = ckind(B', c) = \texttt{lub}}{(B; \cdot c) \sim (B'; \cdot c)}\;[\textsc{s-l0}]$$

$$\frac{ckind(B, c) = ckind(B', c) = \mathsf{box}}{(B; \{c\}) \sim (B'; \{c\})}\;[\textsc{s-l0}] \qquad \frac{ckind(B, c) = \mathsf{hub} \quad ckind(B', c) = \mathsf{box}}{(B; c\cdot) \sim (B'; \{c\})}\;[\textsc{s-hb0}]$$

$$\frac{ckind(B, c) = \mathsf{hub} \quad ckind(B', c) = \texttt{lub}}{(B; c\cdot) \sim (B'; \cdot c)}\;[\textsc{s-hl0}] \qquad \frac{ckind(B, c) = \texttt{lub} \quad ckind(B', c) = \mathsf{box}}{(B; \cdot c) \sim (B'; \{c\})}\;[\textsc{s-lb0}]$$

$$\frac{\begin{array}{c}(B; w) \sim (B'; w') \\ ckind(B, c) = ckind(B', c) = \mathsf{box}\end{array}}{(B; \{c, w\}) \sim (B'; \{c, w'\})}\;[\textsc{s-b1}] \qquad \frac{\begin{array}{c}(B; v) \sim (B'; v') \\ ckind(B, c) = ckind(B', c) = \mathsf{hub}\end{array}}{(B; c\cdot v) \sim (B'; c\cdot v')}\;[\textsc{s-h1}]$$

$$\frac{\begin{array}{c}(B; l) \sim (B'; l') \\ ckind(B, c) = ckind(B', c) = \texttt{lub}\end{array}}{(B; l\cdot c) \sim (B'; l'\cdot c)}\;[\textsc{s-l1}] \qquad \frac{\begin{array}{c}(B; v) \sim (B'; w) \\ ckind(B, c) = \mathsf{hub} \quad ckind(B', c) = \mathsf{box}\end{array}}{(B; c\cdot v) \sim (B'; \{c, w\})}\;[\textsc{s-hb1}]$$

$$\frac{\begin{array}{c}ckind(B', c) = \texttt{lub} \\ ckind(B, c) = \mathsf{hub} \quad (B; v) \sim (B'; l)\end{array}}{(B; c\cdot v) \sim (B'; l\cdot c)}\;[\textsc{s-hl1}] \qquad \frac{\begin{array}{c}ckind(B', c) = \texttt{lub} \\ ckind(B, c) = \mathsf{box} \quad (B; w) \sim (B'; l)\end{array}}{(B; \{c, w\}) \sim (B'; l\cdot c)}\;[\textsc{s-bl1}]$$

Expressions $\boxed{(B; e) \sim (B'; e')}$

$$\frac{}{(B; x) \sim (B'; x)}\;[\textsc{s-var}] \qquad \frac{(B; e_1) \sim (B'; e'_1) \quad (B; e_2) \sim (B'; e'_2)}{(B; (e_1, e_2)) \sim (B'; (e'_1, e'_2))}\;[\textsc{s-tup}]$$

$$\frac{(B; e) \sim (B'; e')}{(B; \lambda x.e) \sim (B'; \lambda x.e')}\;[\textsc{s-lam}] \qquad \frac{(B; e) \sim (B'; e')}{(B; \texttt{fix}\ f\ x = e) \sim (B'; \texttt{fix}\ f\ x = e')}\;[\textsc{s-fix}]$$

$$\frac{(B'; e') \sim (B; e)}{(B; e) \sim (B'; e')}\;[\textsc{s-sym}] \qquad \frac{}{(B; \underline{c}) \sim (B'; \underline{c})}\;[\textsc{s-c0}]$$

$$\frac{(B; e_1) \sim (B'; e'_1) \quad (B; e_2) \sim (B'; e'_2)}{(B; e_1\ e_2) \sim (B'; e'_1\ e'_2)}\;[\textsc{s-app}] \qquad \frac{(B; e) \sim (B'; e')}{(B; \#i\ e) \sim (B'; \#i\ e')}\;[\textsc{s-sel}]$$

$$\frac{(B; e_1) \sim (B'; e'_1) \quad (B; e_2) \sim (B'; e'_2)}{(B; \texttt{let}\ x = e_1\ \texttt{in}\ e_2) \sim (B'; \texttt{let}\ x = e'_1\ \texttt{in}\ e'_2)}\;[\textsc{s-let}] \qquad \frac{(B; e) \sim (B'; e')}{(B; c\ e) \sim (B'; c\ e')}\;[\textsc{s-c1}]$$

$$\frac{(B; e) \sim (B'; e') \quad (B; e_1) \sim (B'; e'_1) \quad \ldots \quad (B; e_n) \sim (B'; e'_n)}{(B; \mathsf{case}\ e\ \mathsf{of}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n) \sim (B'; \mathsf{case}\ e'\ \mathsf{of}\ p_1 \Rightarrow e'_1 \mid \ldots \mid p_n \Rightarrow e'_n)}\;[\textsc{s-case}]$$

Fig. 4. Boxity simulation.

Again, the proof of the above proposition is based on induction on the typing derivation of e.

An alternative proof strategy, which may lead to simpler proofs in case of an extended set of boxities, may be to demonstrate that any well-formed boxity decision leads to results that are equivalent to results obtained using a canonical (e.g., boxed) representation for all types.

3 Inference

We now present an inference algorithm that infers an effective well-formed boxity environment from a data-type declaration. Before presenting the effective inference algorithm, we note again that it is straightforward to find a well-formed, but not necessarily effective, boxity environment from a data-type declaration. To do so, we can assign the boxity `box` to all data-types, as suggested in Section 2.2.

For doing better, rule [E-DD] from Section 2.3 suggests a simple algorithm for possibly finding a more effective boxity environment for a data-type declaration. The algorithm works by first constructing an optimistic boxity environment, based on the number of constructors for the involved data types, their arities in the involved constructor environments, the type boxity of constructor arguments, and knowledge about the boxities of previously resolved data types. The algorithm then checks that the optimistic bet is also a well-formed solution, in which case the algorithm returns the optimistic boxity environment. Otherwise, a fully boxed boxity-environment is returned.

As an example where the algorithm initially finds an optimistic bet that is not well-formed, consider the following data type declaration:

```
datatype t = A of s | B of t*t | C
     and s = D of t | E
```

Choosing the optimistic bet with `t : hub` and `s : lub` does not lead to a well-formed boxity environment. The possible cycle through constructed values involving `A` and `D` must somehow involve boxing. In this case, the algorithm results in a fully boxed boxity-environment. Instead, consider the following small change in the declaration:

```
datatype t = A of s | B of t*t | C
     and s = D of {v:t} | E
```

In this case, the optimistic bet is well-formed because the argument to `D` is boxed (`{v:t}` represents a singleton boxed record with label `v` and with type `t`).

Whereas this suggested inference algorithm is very simple, in practice, it returns effective boxity environments for most of the data-type declarations we have found in realistic code bases, including the benchmark programs we shall discuss in Section 6 and the MLKit and MLton compilers. It is also straightforward to demonstrate that the algorithm results in well-formed boxity environments. As we shall discuss in Section 4, in practice, better results can be obtained by extending the notion of boxities slightly, to deal explicitly with enumeration data-type bindings and with singleton data-type bindings.

4 Implementation

As part of the work reported in this paper, double-ended bit-stealing for algebraic data-types has been implemented in the MLKit compiler where it is enabled by default. It is possible to use the classic boxity strategy by passing the flag `-no_high_pointer_tagging` to the `mlkit` compiler executable (we shall later call this configuration for MLKit†, where † means old). It is also possible to have the MLKit report on boxity decisions by passing the flag `-report_boxities` to the compiler (or REPL). The MLKit is freely available from the MLKit GitHub repository. It is also available from the supplied artifact as described in Section 6.5.

The concrete implementation extends the presented type system and inference algorithm only in minor ways. In the concrete implementation, as touched upon in Section 3, enumerations (i.e., data-type bindings with only nullary constructors) and singleton data-type bindings (i.e., data-type bindings that have precisely one unary constructor) are assigned special boxities.

Singleton data-types are sometimes used by programmers for a kind of safe programming to make explicit that particular values serve a particular purpose and to have the type system distinguish such values from other values. In principle, it is straightforward to eliminate singleton data-type bindings that are non-recursive in the front-end of a compiler and having them incur no runtime overhead whatsoever, which means that we can safely avoid a special treatment of such data-type bindings for the purpose of proving soundness and equivalence (i.e., simulation), as demonstrated in Section 2. Eliminating singleton data-type bindings early, however, may have implementation consequences for other important features such as pretty-printing of values and error messages, incremental compilation, and more. Moreover, such an elimination is not possible for recursive singleton data-type binding, such as the singleton data-type binding defined by Okasaki for binomial trees [Okasaki 1999].[4] Thus, for implementation purposes, we introduce the notion of a boxity `single` κ, where κ is itself a boxity. The aims are to support that singleton data-type bindings are represented unboxed and to ensure that when determining the boxity of a type with precisely one single unary constructor, the boxity of the type can be determined to be the boxity of the constructor argument type. For instance, consider the following example:

```
datatype s = A of t * t | B
     and t = S of s * s
     and u = U of t | K of s
```

Here we can assign `s` to have boxity `lub`, `t` to have boxity `single box`, and `u` to have boxity `hub`, which form a well-formed boxity environment. Without use of the `single` κ boxity, we would be forced to assign the boxity `box` to all three type bindings, as we could not know that the type `t` would not need any bits to distinguish its constructors. We also extend the notion of boxities to include the boxity `enum` for describing data-type bindings with only nullary constructors. By including this boxity, the algorithm can easily be extended to guarantee that enumerations are always represented unboxed (using only low-bit tagging) even if they are declared using a mutually-recursive declaration.

The MLKit compiler is based on a system for incremental recompilation and inter-module optimisation [Elsman 2008], which allows for libraries to be compiled in isolation while compile-time information may be passed across compilation boundaries. Information that may pass across compilation boundaries potentially includes definitions of small functions (cross-module inlining), region- and effect-annotated type schemes, and inferred boxities for data-type declarations.

The Standard ML language does not allow programmers to annotate abstract types and type parameters (i.e., type variables) with representation kinds, which could be useful for giving guarantees that a particular type is represented according to a particular boxity scheme [Eisenberg and Peyton Jones 2017]. Instead the programmer is left with the compiler-flags mentioned above and a set of clear rules that can be used to ensure a particular boxity of a type:

(1) Data-type declarations (possibly a group of mutually recursive bindings) are analysed in isolation and may utilise boxity decisions of dependent data-type declarations (all abstract

[4]Okasaki defines binomial trees using a data-type binding `datatype tree = Node of int * Elem.t * tree list`, where the integer in each tree node equals the depth of the tree and where `Elem.t` is the type of the elements stored in each node.

data types are eliminated at compile time through the principle of static interpretation of modules [Elsman 1999]).[5]

(2) Types declared with no unary constructors (i.e., enumerations) are always assigned boxity `enum` and represented unboxed.
(3) Types declared with one unary constructor and any number of nullary constructors may be assigned boxity `lub` if (*) the argument type of the constructor has boxity `box`.
(4) Types declared with at most 2^H constructors (where $H = 16$ is the number of available bits for high-bit tagging) and at least one unary constructor may be assigned boxity `hub` if (*) each unary constructor argument type has boxity `lub` or `box`.
(5) Types declared with precisely one unary constructor may be assigned boxity `single` κ where κ is the boxity of the argument type of the constructor.
(6) In a data-type declaration, if any type-binding fails the conditions (*) in rule 3 or rule 4, then all non-enumeration data-types (i.e., those mentioned in rules 3–5) are assigned boxity `box`.

There is one trick, in particular, that a programmer may apply to improve the boxity of a data-type binding that is inferred to have boxity `box`. A common reason that a data-type binding fails to satisfy condition (*) in rule 4 is that only a few constructors fail the condition. In such cases, a programmer may explicitly box the argument for a constructor by wrapping the argument in a singleton record, which MLKit will ensure to implement boxed. As an example, consider what would happen if we add the constructor "`Int of int`" to the data-type declaration for `e` in Section 1. Because the type `int` has boxity `hub`, the condition (*) in rule 4 above would be violated leading to a boxed representation of `e`. We could instead choose the following addition:

```
datatype e = ... | Int of {int:int}
```

With this additional constructor, which now takes a singleton record as argument (holding an element with label `int` and of type `int`), the data-type binding for `e` would again be assigned the boxity `hub`, as singleton records use a boxed representation. Notice also that the MLKit does not currently attempt to split mutually recursively declared data-type declarations into strongly-connected components, which could potentially improve unboxity decisions.

The curious and observant reader may ask why the MLKit is not always using an unboxed representation for data-type bindings with precisely one unary constructor (it may fail to do so due to rule (6) above). The reason is somewhat arbitrary and is due to the particular region-inference algorithm used in the MLKit, which associates region- and effect-variables with a data-type binding. The algorithm assumes that either all data-type bindings, in a data-type declaration, are boxed (in which case they share a common region) or they are unboxed (they share no common region). We consider it future work to generalise the implementation to guarantee that such constructors are always represented unboxed.

5 Examples

We now present a series of real-world examples that benefit from double-ended bit-stealing.

5.1 Patricia Trees

Patricia trees [Morrison 1968] turn out to be particular useful for representing fast mergeable finite maps with integer domains [Okasaki and Gill 1998]. Here is the essential data structure for representing Patricia trees:

[5]Instead of relying on a mechanism for eliminating abstract data types at compile time, an implementation may, as an alternative, rely on specified representation kinds for abstract types to resolve boxities [Eisenberg and Peyton Jones 2017].

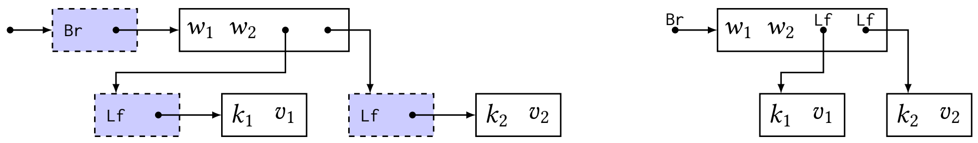

Fig. 5. Boxed and unboxed representations of a Patricia tree Br(w_1,w_2,Lf(k_1,v_1),Lf(k_2,v_2)) as implemented in MLKit† and MLKit. The light-blue (and dashed) rectangular boxes represent boxed constructed values and the white rectangular boxes represent tuples.

```
datatype α map = Empty
               | Lf of word * α
               | Br of word * word * α map * α map
```

For branch nodes, the two stored word values specify the *branching bit*, the bit specifying whether a node is potentially to the left or to the right in the tree, and the largest common prefix for all the keys in the tree. Keys and values are stored at leaf notes and the `Empty` constructor specifies the empty map. Figure 5 shows a boxed and an unboxed representation of a simple Patricia tree as they may be implemented in MLKit† and MLKit. Looking at the layouts in the figure, it is clear that saving the intermediate constructor values decreases pointer-tracing, memory usage, and code related to memory allocation and deallocation (e.g., garbage collection). In Section 6, we shall return to the performance benefits of double-ended bit-stealing for Patricia trees (and for the other examples in this section).

Notice that using multi-constructor arguments, which is how many ML implementations represent the `map` data type, the memory usage is only slightly higher than when the MLKit representation is used (one additional word for each block). Notice also that in comparison to many other tree structures, including AVL-trees and red-black trees, Patricia trees are special in that they have two unary constructors. In comparison, AVL-trees and red-black trees may be implemented unboxed in MLKit using low-bit unboxing, which is not possible with Patricia trees.

5.2 Union-Find Data Structures

A union-find imperative data-type is essential for many purposes, in particular for type inference and other program-analyses, including, for instance, region-inference [Tofte and Birkedal 1998]. A union-find data structure can be hidden behind an interface for *unifiable references*, a concept that allows for mutable cells to be unioned, after which the cells become indistinguishable. The underlying union-find data structure may be defined using the following Standard ML data-type declaration:

```
datatype α t0 = ECR of α * int | PTR of α t
withtype α t = α t0 ref
```

Figure 6 shows a boxed and an unboxed representation of a value of type `data t`, where `data` is the type of data carried by the union-find structure. Central to all operations on union-find structures is an operation for extracting the equivalence class representative (ECR) of the structure, while compressing or halving the pointer paths introduced by union operations. While we shall not discuss the best way to do so, it is clear that no matter which way is used (path-compression or path-halving) and no matter whether union operations take into account the *rank* of a node (i.e., how many `PTR` values point to an `ECR` node), it is essential to minimise the memory occupancy (e.g., for cache purposes) and the number of indirections introduced in the data structure. It is

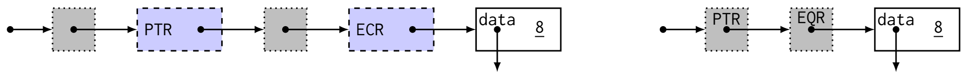

Fig. 6. Boxed and unboxed representations of the union-find structure value `ref(PTR(ref(ECR(data,8))))` as implemented in MLKit† and MLKit. Singleton gray (and dotted) boxes represent `ref` cells, the light-blue (and dashed) rectangular boxes represent boxed constructed values, and the white rectangular boxes represent pairs.

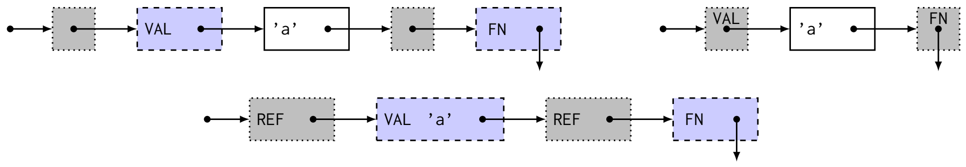

Fig. 7. Three different uniform representations of a value of type `char stream`. The top-left representation is how the value is represented in MLKit† (without unboxing) whereas the top-right representation is how the value is represented unboxed in MLKit using high-bit pointer tagging. The representation below is how the value is traditionally represented in ML implementations such as MosML, OCaml, and SML/NJ. Gray (and dotted) boxes represent `ref` cells, the light-blue (and dashed) rectangular boxes represent boxed constructed values, and the white rectangular boxes represent pairs.

also clear that the representation to the right in Figure 6 is much more compact than the boxed representation to the left in the figure. We shall later, in Section 6, investigate the performance benefits of the more compact representation.

Notice that some ML implementations (e.g., MosML, OCaml, and SML/NJ), compared to the boxed MLKit† representation, use a multi-argument representation of `ECR` nodes, while the `ref` cells use an additional word for tagging.

5.3 ML-Yacc and ML-Lex Parsing and Lexing

Lexing and parsing are central to compilers and time spend lexing and parsing can often constitute large percentages of overall compilation times. The ML-Lex and ML-Yacc lexing and parsing generators generate code that makes use of a series of data structures, including lazy streams and semantic values that benefit from unboxing. Lazy streams are represented using the following data-type declaration [Tarditi and Appel 2000]:

```
datatype α str = VAL of α * α stream | FN of unit → α
withtype α stream = α str ref
```

Using this data structure, it can be guaranteed that elements are fetched from the underlying source only once. Notice that the `stream` type is imperative and that the data type supports arbitrary look-ahead. Figure 7 shows three different unboxing schemes for a `char stream` value of the form

```
val s : char stream = ref (VAL(#"a",ref(FN f)))
```

Here we assume that `f` is a function (of type `unit → char`) that reads consecutive characters from a file or a buffer and returns the special character EOF when there are no more proper characters to be read.

Another ML-Yacc data type that is a candidate for unboxing is the data type for ML-Yacc semantic values. This data type is generated by ML-Yacc for every grammar and is part of what constitutes a token value. Semantic values are tagged functions and for the Standard ML grammar, the data type has 142 unary constructors, taking functions as arguments, and one nullary constructor. For the `calc` benchmark presented in Section 6, the grammar for arithmetic expressions leads to a data type for semantic values with five unary constructors, taking functions as arguments, and one nullary constructor. In both cases, MLKit will use an unboxed representation for the data type.

5.4 Alternative Representations of the `option` Type

In the MLKit, the built-in `option` type is represented boxed. Here is an alternative specification of the `option` type, which may be represented using low-unboxing (`lub`) according to the scheme in this paper:

```
datatype α option = NONE | SOME of {v:α}
```

Here we use the property that singleton records are always boxed, as hinted upon in Section 4.

For values of type `'a option option`, we could potentially choose another representation for the outer `option` (using high-unboxing, with no associated allocation), but for uniformity reasons, the same representation is chosen for both instances of the `option` type.

5.5 Large Abstract Syntax Trees

The double-ended bit-stealing technique is useful for obtaining unboxed representations for many recursively defined term data types, including, for instance, term structures for intermediate-language type structures and intermediate-language data types for statements and expressions in compilers. Many of these data types resemble the introductory example for expressions in Section 1.

Concretely, for the MLKit compiler, here is a non-exhaustive list of term data types that allow for high-bit unboxing:

`TopdecGrammar` types. These types include 23 mutually recursive data type declarations that are all inferred simultaneously to be candidates for high-bit unboxing. Among others, the types include variants for expressions, atomic expressions, patterns, atomic patterns, declarations, module declarations, and signature expressions. The types are either singleton data types (data types with only one unary constructor and no nullary constructors) or data types with constructors that take only boxed or low-unboxed arguments.

`LambdaExp` terms for types and expressions. `LambdaExp` is an explicitly typed intermediate language on which a large number or repeated optimisations are performed. Terms of `LambdaExp` include both type expressions and a single data type for language expressions. Both data types are inferred to be candidates for high-bit unboxing. `LambdaExp` type expressions do not support unification but are the result of translating elaboration types that have been attached to the `TopdecGrammar` structure during elaboration [Milner et al. 1997].

`RegionExp` terms for types and expressions. `RegionExp` is an explicitly region- and effect-typed intermediate language. It is the target language of region inference and is used both for the so-called spreading phase and for the so-called fix-point phase of region inference [Tofte and Birkedal 1998]. `RegionExp` expressions are polymorphic both in annotated allocation information and in annotated region- and effect-binding information, yet, both the data type for `RegionExp` types and the data type for `RegionExp` expressions are inferred to be candidates for high-bit unboxing. `RegionExp` types form a type structure where types for allocated values are annotated with regions and where function types are annotated with effects. `RegionExp` types support a notion of unification where annotated regions and effects in types may be unified using an underlying union-find data structure. The key components of this data structure are also candidates for high-bit unboxing, as described in Section 5.2.

`MulExp` and `ClosExp` terms for expressions. These term structures define internal language representations in the form of data type declarations for so-called multiplicity-inferred expressions [Birkedal et al. 1996] and for so-called closure-explicit expressions, which are both refined versions of `RegionExp` expressions. Both data types are candidates for high-bit unboxing and as for `RegionExp` terms, `MulExp` terms and `ClosExp` terms are polymorphic both in annotated allocation information and in annotated region- and effect-binding information.

`LineStmt` terms for expressions and statements. Data types for these terms define an SSA-like intermediate language on which to perform low-level optimisations, register allocation, and instruction selection for particular architectures. both the data type for `LineStmt` expressions and the data type for `LineStmt` statements are inferred to be candidates for high-bit unboxing.

We emphasise here that unboxing of the above data types complements the unboxing of Patricia-trees, union-find structures, effect graphs, and involved data types for lexing and parsing, as described in Section 5, along with a series of other supportive data types, such as data types for pretty printing and more. We shall discuss in Section 6.3 how double-ended bit-stealing affects the performance of the MLKit compiler.

6 Benchmarks

In this section, we report on the consequences of the optimised data type representation for a series of benchmarks programs. We compare runtimes and memory usage for the MLKit configuration that uses only boxed and low-unboxed data type representations (MLKit†), with a configuration that uses boxed, high-unboxed, and low-unboxed representations (MLKit). The MLKit† configuration resembles the preexisting MLKit boxity technique (hence the use of †). Measurements are performed with MLKit v4.7.11. We also compare the runtime and memory usage with measurements obtained with a configuration of MLKit that combines high-bit unboxing with region inference for memory management, but no complementary reference-tracing garbage collection (MLKit$^\mathrm{R}$). MLKit$^\mathrm{R}$ uses full 64-bit (untagged) integers and all records and tuples are allocated without header tags (no garbage collector will need to traverse values at runtime); it is the version of MLKit with the least mutator overhead. Further, we compare the runtime and memory usage with measurements obtained with MLton 20210117 [Weeks 2006], a whole-program optimising Standard ML compiler, and SML/NJ 110.99.4, a Standard ML compiler based on uniform data representations and a copying reference-tracing and generational garbage collector [Appel 1992; Appel and MacQueen 1987]. The comparison with MLton and SML/NJ serves to relate the performance of the code generated by MLKit with the performance of two state-of-the-art compilers for Standard ML. We emphasise that MLKit, SML/NJ, and MLton are three very different compilers.

All benchmark programs are executed on a MacBook Pro (15-inch, 2016) with a 2.7GHz Intel Core i7 processor and 16GB of memory. Times reported are wall clock times and memory usage is measured using the GNU `time` program.

The benchmark programs span from micro-benchmarks that serve to demonstrate benefits of the optimised unboxed data type representation on particular data types to larger benchmarks that make essential use of data types that are candidates for high-bit unboxing. Benchmarks that fall in the former category are the benchmarks `uf` (see Section 5.2) and `patricia` (see Section 5.1), where the remaining benchmarks fall in the latter category. The `calc` benchmark is a calculator program, constructed using ML-Lex and ML-Yacc, which calculates the value of large arithmetic expressions written as strings that are lexed and parsed by generated lexers and parsers. The benchmark makes

		Real time (s)					
Program	**Lines**	**MLton**	**SML/NJ**	**MLKit$^{\text{R}}$**	**MLKit†**	**MLKit**	**Speedup**
`calc`	1657	2.25 ± 0%	**1.62 ± 1%**	2.13 ± 1%	2.45 ± 2%	2.37 ± 0%	3.4%
`DLX`	2497	**0.05** ± 7%	3.88 ± 1%	0.60 ± 1%	0.42 ± 2%	0.41 ± 3%	2.4%
`kbc`	639	**0.27** ± 2%	0.51 ± 2%	0.59 ± 2%	0.75 ± 1%	0.74 ± 4%	1.4%
`lexgen`	1326	**0.39** ± 1%	0.63 ± 0%	0.55 ± 0%	0.64 ± 1%	0.62 ± 1%	3.2%
`logic`	360	**0.65 ± 1%**	0.81 ± 1%	1.08 ± 1%	1.34 ± 1%	1.14 ± 1%	17.5%
`nucleic`	3219	0.83 ± 1%	**0.45 ± 1%**	1.44 ± 1%	1.16 ± 1%	1.16 ± 2%	0%
`patricia`	292	**3.08 ± 1%**	4.70 ± 1%	2.31 ± 3%	4.73 ± 1%	3.82 ± 0%	23.8%
`ray`	544	**0.24** ± 0%	0.27 ± 2%	0.53 ± 1%	0.62 ± 3%	0.60 ± 2%	3.3%
`uf`	83	**0.14** ± 0%	0.90 ± 2%	0.31 ± 2%	0.48 ± 0%	0.38 ± 2%	26.3%
`vliw`	3681	**0.29** ± 0%	0.48 ± 4%	0.44 ± 1%	0.56 ± 3%	0.55 ± 1%	1.8%

Fig. 8. Average wall-clock execution times, measured in seconds, of 10 consecutive runs of the benchmark programs. The speedup column reports the speedup of the MLKit column compared to the MLKit† column.

essential use of the `stream` data type presented in Section 5.3 and allows also for high-bit unboxing for so-called semantic values, as outlined also in Section 5.3. The `lexgen` benchmark repeats 20 times the task of generating a lexer for Standard ML, based on an ML-Lex specification. The `dlx` benchmark is a partial interpreter for Patterson's and Hennesey's RISK computer architecture running the Euclidean gcd algorithm 20 times on the numbers 2322 and 654. The benchmark allows for high-bit unboxing of the representation of RISK instructions. The `kbc` and `logic` benchmarks perform Knuth-Bendix completion and find solutions to logical problems. Both benchmarks allow for high-bit unboxing for the respective involved data types for terms. The `ray` benchmark performs ray-tracing on a 3D object model and allows for high-bit unboxing of the recursive bounding box structure. The `nucleic` benchmark computes the length of an anti-codon structure 200 times. It allows for high-bit unboxing of the internal data type for atoms. The `uf` and `patricia` benchmarks are described in Section 5.2 and Section 5.1, respectively.

6.1 Execution Time

Figure 8 lists the benchmark programs and reports, for each benchmark, line counts and execution times across compiler configurations. Measurements are averages over 10 runs. The **Real time** columns list average execution times in seconds, annotated with relative standard deviations.

There are several observations to be made. The best time-performing numbers (in the respective rows) are marked with **bold** whereas significant improvements from high-bit unboxing (MLKit compared with MLKit†) are underlined. From an absolute perspective, we see that SML/NJ is fastest for the `nucleic` and `calc` benchmarks and that MLton is fastest for the majority of benchmarks. We note here that both MLton and SML/NJ benefit from effective flattening techniques [Ziarek et al. 2008] that allow for specialised flattened records of real values and argument flattening that effectively allow functions to receive real values in floating point registers; such optimisations have not yet been implemented to the same extend in MLKit. We also emphasise that MLton, SML/NJ, and MLKit are three very different compilers that are based on different compilation strategies. We see that the whole-program compilation scheme used by MLton leads to superior code in most cases and in particular for the `DLX` benchmark, which benefit from MLton's superior inlining and handling of 32-bit word-operations. While MLKit represents 32-bit word values as boxed values, we cannot explain the extraordinary high runtime for SML/NJ in the case of the `DLX` benchmark. We note here that the whole-program compilation scheme of MLton comes with a cost. In particular, whole-program compilation schemes suffer from slow recompilation for large code bases.

		Memory usage (MiB ± 0 − 2%)					
Program	**Lines**	**MLton**	**SML/NJ**	**MLKit^R**	**MLKit†**	**MLKit**	**Decrease**
calc	1657	68.2	44.9	369	81.5	75.2	7.7%
DLX	2497	9.5	16.9	60.7	18.1	18.1	0%
kbc	639	2.3	5.8	5.4	6.6	5.6	15.1%
lexgen	1326	18.4	7.5	67.0	21.4	18.3	14.5%
logic	360	4.9	6.0	852	3.6	3.4	5.6%
nucleic	3219	2.0	5.0	1158	5.2	5.2	0%
patricia	292	15.2	14.8	6.7	7.7	6.2	19.5%
ray	544	14.2	9.5	12.2	9.1	8.5	6.6%
uf	83	27.3	43.6	4.8	6.7	5.1	23.9%
vliw	3681	9.2	9.9	42.9	16.2	12.9	20.4%

Fig. 9. Average memory usage, measured in MiB (megabytes), for 10 consecutive runs of the benchmark programs. The different columns show results for the different compilers and configurations. The standard deviations for the measurements are below 2 percent. The last column reports the decrease in memory usage (in percent) of the MLKit column compared to the MLKit† column.

None of the benchmarks results in slower code with MLKit than with MLKit†. Speedup (MLKit† / MLKit) ranges from 0 to 26 percent, with the `uf` and `patricia` benchmarks showing speedups of 23 and 26 percent, respectively. The `logic` benchmark shows 17 percent speedup. For the remaining benchmarks, the critical code paths do only to a limited extend involve constructing and deconstructing values that are represented using double-ended bit-stealing. Notice, however, that double-ended bit-stealing never results in slowdowns.

For the MLKit^R configuration, we see that execution times are often faster than the other MLKit configurations, except for the `nucleic` benchmark, which, as we shall discuss in the next section, in essential ways depends on complementing region inference with reference-tracing garbage collection.

6.2 Memory Usage

Average memory usage, measured in MiB (megabytes), for 10 consecutive runs of the benchmark programs are shown in Figure 9. The maximum memory usage across compiler configurations, with the exception of the MLKit^R compiler configuration is about 82 MiB. The numbers vary across configurations for each particular configuration, which reflect the different nature of the memory management schemes used and the different default memory management settings used (e.g., the default heap-to-live ratios used). For the benchmark programs `patricia` and `uf`, we see some evidence of the more compact data type representation in MLKit compared to MLKit†.

With respect to the MLKit^R configuration, we see that some of the benchmarks, in particular `calc`, `logic`, and `nucleic`, demonstrate problematic memory behavior due to the lack of a reference-tracing garbage collector to complement region inference. On the other hand, the memory behavior of many of the other benchmark programs is fine. In particular, we see that the memory usage for the benchmarks `kbc`, `patricia`, and `uf` is comparable to the memory usage for the configurations that use reference-tracing garbage collection. For the `nucleic` benchmark program, the memory usage is so high that it drastically influences the running time of the program, as noted in the previous section.

		Run time (s)			Memory usage (MiB)		
Program	**Lines**	**MLKit†**	**MLKit**	**Speedup**	**MLKit†**	**MLKit**	**Decrease**
`MLKit`	103857	488 ± 0.1%	445 ± 0.3%	9.7%	950 ± 0.0%	758 ± 0.0%	20.2%
`MLton`	198487	2012 ± 0.8%	1838 ± 0.1%	9.5%	4831 ± 0.0%	4276 ± 0.0%	11.5%

Fig. 10. Average wall-clock execution times and average memory usage for compiling MLKit and MLton with different configurations of MLKit. Average wall-clock times are measured in seconds and average memory usage is measured in MiB (megabytes). Numbers are averages of 5 consecutive runs. The relative standard deviations for the measurements are below 2 percent.

	Algebraic Data Types And Their Boxities (κ)								
Program/Compiler	**Tot**	`hub`	`lub`	`enum`	`box`	`single` κ'	κ' = `hub`	κ' = `lub`	κ' = `box`
`MLKit` / MLKit†	434	0	52	72	227	83	6	15	62
`MLKit` / MLKit	434	127	52	72	45	138	7	16	115
`MLton` / MLKit†	714	0	51	116	254	293	8	11	274
`MLton` / MLKit	714	129	48	116	112	309	8	11	257

Fig. 11. Data type boxity counts for compiling MLKit and MLton with the MLKit† and MLKit configurations of the MLKit compiler.

6.3 Comparison of Bootstrapped Compilers

In this section, we report on the effect of double-ended bit-stealing in the MLKit Standard ML compiler when used for compiling the MLKit itself and for compiling MLton. Double-ended bit-stealing applies to a large series of supporting data types, used by MLKit, as described in Section 5 and for many of the intermediate languages of MLKit, as described in Section 5.5.

Figure 10 shows average wall-clock times and memory usage for compiling the MLKit and MLton sources using the MLKit and MLKit† configurations of the MLKit compiler. It is not tractable to compile MLKit and MLton with the MLKit^R configuration due to the lack of a dynamic reference-tracing garbage collector in this configuration.

We can see that double-ended bit-stealing has an overall positive effect on both execution time performance and on memory usage. For compiling MLKit itself, the speedup is 9.7 percent and the decrease in memory usage is 20.2 percent. For compiling MLton, the speedup is 9.5 percent and the decrease in memory usage is 11.5 percent.

6.4 Broadness of Double-Ended Bit-Stealing

To understand better the broad applicability of double-ended bit-stealing, we have counted the different boxity decisions made when the two MLKit compiler configurations MLKit and MLKit† compiles MLKit and MLton, respectively. As described in Section 4, the MLKit defines boxities (κ) according to the following grammar:

$$\kappa ::= \texttt{hub} \mid \texttt{lub} \mid \texttt{box} \mid \texttt{enum} \mid \texttt{single}\ \kappa$$

Here, the `enum` boxity is assigned to data types with only nullary constructors and the `single` κ boxity may be assigned to data types with zero nullary constructors and only a single unary constructor, for which the argument has boxity κ.

Figure 11 reports boxity counts for compiling MLKit and MLton with the MLKit† and MLKit configurations of the MLKit compiler. We see that MLKit declares 434 data types of which 72 are enumerations (boxity `enum`) and 52 are low-unboxed types (boxity `lub`), which hold true for both the MLKit† and MLKit configurations of the compiler. For compiling MLKit with the MLKit† compiler, 227 of the 434 data types are inferred to use a boxed representation whereas this number is only 45 for the MLKit configuration of the compiler. On the other hand, with the MLKit configuration, 127 data types are inferred to be unboxed using high-bit tagging.

For boxities of the form `single` κ', we see a higher count for the MLKit configuration than for the MLKit† configuration of the compiler. The reason is that when high-bit tagging is disabled for data types (when only basic types such as `int` and `word` are considered high-unboxed and have boxity `hub`) inference has conservatively determined that some singleton algebraic data types should use a boxed representation, as discussed in Section 4. Future work may refine the decision procedure to avoid always a boxed representation of singleton data types.

For MLton, we see that the implementation declares a total of 714 data types of which 116 are enumerations (boxity `enum`). For MLKit†, 254 data types use a boxed representation whereas this number is 112 when high-bit unboxing is enabled, in which case 129 data types are represented using high-bit unboxing. We also see that there is a difference with respect to the number of data types that use a low-bit untagged representation, which can happen as the boxity of a constructor argument may influence the boxity of the data type, as described in Section 3.

Figure 11 lists also numbers for three possibilities of κ' for boxities of the form `single` κ'. For the first three rows, the numbers for the three columns sum up to the number in the `single` κ' column, which is not the case for the last row (`MLton` / MLKit case). The reason is that the inference algorithm sometimes finds nested singleton boxities of the form `single` (`single` κ), which will be promoted to boxity `box` by MLKit† (which also explains the difference between the two numbers in the `single` κ column for the `MLton` cases).

6.5 Artifact

The paper comes with an artifact that includes a tutorial demonstrating unboxing in the MLKit (and in particular unboxing through double-ended bit stealing) as described in the paper, code for replicating the benchmark results (Figure 8 and Figure 9), code for replicating the results of compiling MLton and MLKit itself using different configurations of MLKit (Figure 10 and Figure 11), and, finally, the source code for MLKit 4.7.11, including a description of the source code implementation aspects of double-ended bit-stealing. The artifact is available both on GitHub in source form[6] and as a Zenodo image [Elsman 2024].

7 Related Work

Related work falls into a series of categories. Much related to our work is the recent work on reasoning about the soundness of bit-stealing in RIBBIT [Baudon et al. 2022, 2023]. This work allows for specifying exactly the use of bits in a machine word and for specifying, in particular, how different variants of a data type are discriminated in a safe way. Examples include a specification of how so-called NaN-boxing[7] may be used for discriminating between values in a JavaScript engine and specifications of how pointer-tagging can be used for specifying unboxed representations of lists. There is no reason that the techniques used should not also work for double-ended bit-stealing, although no examples are given utilising the higher unused bits of pointer values. Another difference from our work is that, compared to RIBBIT, which serves as a specification language for data-layouts, the double-ended bit-stealing technique that we present aims at inferring unboxed representations based on a set of simple rules, while ensuring that the inferred representations work in a polymorphic setting. This line of work also includes the work on so-called niches in

[6]Artifact GitHub URL: https://github.com/melsman/debs-icfp24.

[7]NaN-boxing refers to a technique for encoding the union of all IEEE float values and all 64-bit valid pointers into one 64-bit machine word, while still allowing all values to be distinguished.

Rust, which allow for using alignment bits (e.g., lower bits in aligned pointers) in a safe way [Rust Community 2021].

Also related to this work is the work on using kinds for distinguishing different layouts and calling conventions [Downen et al. 2020; Eisenberg et al. 2022; Eisenberg and Peyton Jones 2017], which originates from the work on TALT [Crary 2003] and Cyclone [Jim et al. 2002], which use kinds to classify different values according to the number of bytes they are represented by at runtime. None of these systems support the use of the unused high-bits in pointers for value tagging. The implementation of double-ended bit-stealing in the MLKit does not give programmers the ability to specify the boxity of a type and therefore, programmers are not guaranteed that the boxity properties obtained are satisfied after changes to source code (or after changes to the compiler). It would be natural to add a source-level kind system (or an attribute system similar to that of OCaml 5.1) that would allow programmers to specify representation properties, such as boxities, for abstract types and type variables.

Another interesting approach to unboxing (and untagging) is to identify algebraic data types for which it is possible to discriminate between variants (different constructed values) based on the values carried in the payload of the constructors [Chataing et al. 2024]. An example is a data type declaration that declares two constructors, one that takes an integer as argument and one that takes a string as argument. Because it is possible to discriminate between integers and strings at runtime, boxing and tagging can be avoided altogether. We consider it future work to investigate how to complement such an unboxing strategy with double-ended bit-stealing.

There is a strand of related work that addresses the task of compactly representing objects while still allowing for separately compiled polymorphic code. This work includes Leroy's seminal paper on Polymorphic typing and unboxing [Leroy 1992], Peyton Jones and Launchbury's work on unboxing in a non-strict functional language [Peyton Jones and Launchbury 1991], the work by Henglein and Jørgensen on formally optimal boxing [Henglein and Jørgensen 1994], and revisions of this work [Thiemann 1995]. This strand of work also includes work on practical representation choices for built-in types in OCaml, and how ad-hoc techniques can be used to obtain unboxed representations of records of floats and float arrays [Leroy 1997]. Whereas many of the techniques introduced in this strand of work are of importance to achieving high-performance, the work is not particularly concerned with the representation of algebraic data-types and is in most respects complementary to the work presented in this paper.

Another line of related work is the work on tuple- and argument-flattening in Manticore [Bergstrom and Reppy 2009] and in the MLton compiler [Ziarek et al. 2008], which supports flattening even of reference cells into records but which also comes at the price of requiring whole-program compilation. MLKit suffers from the lack of being sufficiently good at deep flattening of function arguments and of tuples in data structures. In particular, being able to pass floating point values in registers and being able to store floating point values in records without indirections to separate blocks of memory are essential, but are areas of research that we do not cover in this work.

A general technique for allowing specialised non-uniform representations of values is to pass representations of types to functions at runtime. This line of work includes work on adding a type-case construct to an intermediate language [Harper and Morrisett 1995; Morrisett 1995] and work on implementing polymorphism in ML-like languages using layout bit-maps [Nguyen and Ohori 2006]. Other work in this area includes work on dictionary-passing transformations for implementing type classes [Hall et al. 1996; Jones 1994; Peterson and Jones 1993] and ad-hoc polymorphism [Elsman 1998].

Also related to this work is the concept of generalised algebraic data types (GADTs), which may be used, for instance, for writing type-indexed tag-free interpreters. Whereas such untagging is unrelated to the unboxing techniques presented here, with some restrictions [Colin et al. 2019], there is no reason that the techniques presented in this paper should not carry over to work also for obtaining unboxed representations of generalised algebraic data types. Another technique for achieving untagged implementations of interpreters is the finally tag-less approach, which uses techniques such as existential types and phantom types to avoid tagged variants altogether [Carette et al. 2009].

Another source of related work is work on using fully flattened representations of algebraic data types [Koparkar et al. 2021]. Whereas such representations and techniques are interesting from a data-parallel perspective and from a streaming perspective, in general, fully flattened representations of algebraic data types (i.e., flat pointer-free representations) lack important properties of algebraic data types that are useful for general-purpose computation.

8 Conclusion and Future Work

We have devised a technique for double-ended bit-stealing for algebraic data types, show that the technique is sound, and demonstrated that it leads to improvements of code generated by a compiler for a complete ML language.

There are several opportunities for future work. First, from a language perspective, it would be natural to support a notion of representation polymorphism that allows programmers to be explicit in the code about representation kinds [Eisenberg and Peyton Jones 2017]. In the context of ML-like languages featuring module systems, it would be natural to support representation specifications for abstract data types in module types. It would also be natural to support a notion of restricted parameterised types for supporting that a type variable can be instantiated only to types that have certain representation guarantees. Allowing programmers to specify representation kinds for abstract types supports modular performance in the sense that proper representation guarantees can be given without knowing all details about foreign code.

Second, we also consider it future work to combine double-ended bit-stealing with other opportunities for unboxing, such as relying on discriminating between constructed values based on their payload [Chataing et al. 2024], and with opportunities for representing constructor payloads more compactly than using uniform representations, perhaps by using spare bits in header tags.

Another strand of future work is to implement a flexible tuple- and argument-flattening technique that more closely resembles the techniques used in SML/NJ and MLton. Currently, the tuple-flattening technique used in MLKit is not capable of flattening tuples of tuples, for instance, as discussed in Section 7.

Local specialisations of tight code and the associated data types may also be beneficial for achieving a performance closer to that of MLton. We believe that local specialisations and more flexible tuple-flattening may together make double-ended bit-stealing an even better optimisation technique than demonstrated in this paper.

Data Availability Statement

This paper is accompanied by a software artifact [Elsman 2024] that demonstrates the main contributions of the paper. A more detailed description of the content of the software artifact is available in Section 6.5.

Acknowledgments

We would like to thank the anonymous reviewers for careful and constructive feedback on this work. The work builds in essential ways upon the MLKit compiler infrastructure to which many people have contributed.

A Appendix: Language Properties and Proofs

This appendix contains proof details for the paper "Double-Ended Bit-Stealing for Algebraic Data Types" (ICFP '24) and an overview of notation and syntactic constructs (Section A.3).

A.1 Basic Properties and Soundness

The properties that we state in this section are proven using well-known techniques for proving type safety [Morrisett 1995; Wright and Felleisen 1994].

Proposition 2.2 (Typing Rules Closed Under Type Substitution). *If $B, \Gamma \vdash e : \sigma$ then $B, S(\Gamma) \vdash e : S(\sigma)$, for any substitution S.*

Proof. By straightforward induction over the structure of the typing derivation. The proof makes use of the facts that boxity environments are always closed with respect to type variables and that type scheme instantiation is closed under substitution. Moreover, for any substitution S, we have that $\Gamma = (B, \vec{\tau}\ t) \triangleright p$ implies $S(\Gamma) = S(B, \vec{\tau}\ t) \triangleright p$, for any environment Γ, boxity environment B, type $\vec{\tau}\ t$, and pattern p, which also makes use of the fact that type scheme instantiation is closed under substitution.

For [e-fix], we use the property that type schemes are considered equal up to alpha-renaming and that, by renaming, we can assure that $\{\vec{\alpha}\} \cap \mathrm{Dom}\, S = \emptyset$ and $\{\vec{\alpha}\} \cap \mathrm{ftv}(S(\Gamma)) = \emptyset$.

For rules where environments are extended in the premises (e.g., [l-fix] and [e-lam]), we use the property that $S(\Gamma, x : \sigma) = S(\Gamma), x : S(\sigma)$. □

Proposition 2.3 (Unique Decomposition). *If $B \vdash e : \sigma$ then either e is a value or there exists a unique redex r, a type scheme σ', and a context E such that $B \vdash r : \sigma'$ and $e = E[r]$.*

Proof. By induction over the typing derivation. There are ten non-value cases corresponding to the ten typing rules for which e is not a value or a variable. We show a few cases here.

case [e-inst]. We have $B \vdash e : \sigma'$ and $\sigma = \tau$ and $\sigma' \geq \tau$. Here we assume σ' is a proper type scheme (not just a type), which ensures that the inductively defined typing relation is well-founded (which is standard). The required conclusion then follows directly by induction.

case [e-con1]. We have $\sigma = \vec{\tau}\ t$ and $e = c\ e'$ and $B(t)(c) \geq \tau \rightarrow \vec{\tau}\ t$ and $B \vdash e' : \tau$. By induction, we have either (1) e' is a value or (2) there exists a unique redex r', a type scheme σ'', and a context E' such that $B \vdash r' : \sigma''$ and $e' = E'[r']$. In case of (1), we have $r = e = c\ e'$ is a unique redex (e' is a value). Further, we have $\sigma' = \sigma$ and $E = [\cdot]$ and $e = E[r]$ and $B \vdash r : \sigma'$ (straightforward from assumptions), as required. In case of (2), we have $\sigma' = \sigma''$ and $E = c\ E'$ and $r = r'$ and $e = E[r]$ and therefore $B \vdash r : \sigma'$, as required.

The remaining cases follow similarly. □

Proposition 2.4 (Context). *If $B \vdash E[e'] : \sigma$ then $B \vdash e' : \sigma'$ for some σ'. Further, if $B \vdash E[e] : \sigma$ and $B \vdash e : \sigma'$ and $B \vdash e' : \sigma'$ then $B \vdash E[e'] : \sigma$.*

Proof. Each of the properties are proven by straightforward induction over the structure of E. □

Proposition 2.5 (Value Substitution). *If $B, \Gamma, x : \sigma' \vdash e : \sigma$ and $B \vdash w : \sigma'$ then $B, \Gamma \vdash e[w/x] : \sigma$.*

Proof. By induction over the structure of e using an *environment extension property* of typing stating that if $B, \Gamma \vdash e : \tau$ then $B, \Gamma, \Gamma' \vdash e : \tau$ for any Γ' with $\mathrm{Dom}(\Gamma') \cap \mathrm{Dom}(\Gamma) = \emptyset$. As is standard, the proof also makes use of Barendregt's convention for renaming bound variables in expressions for avoiding environment capture. The interesting case is the case for variables (rule [e-var]) for which $\sigma = \sigma'$. The required conclusion then follows directly from assumptions. □

Proposition 2.6 (Type Preservation). *If* $\vdash B$ *and* $B \vdash e : \sigma$ *and* $B \vdash e \rightsquigarrow e'$ *then* $B \vdash e' : \sigma$.

Proof. By induction over the structure of e. From the small-step reduction rules, there are 20 cases. We show some of the cases here.

case [r-ctx]. We have $B \vdash e_1 \rightsquigarrow e_1'$ and $E \neq [\cdot]$ and $e = E[e_1]$ and $e' = E[e_1']$. By Proposition 2.4, we have there exists σ' such that $B \vdash e_1 : \sigma'$. We can now apply induction to get $B \vdash e_1' : \sigma'$. Finally, by the second property in Proposition 2.4, we have $B \vdash e' : \sigma$, as required.

case [r-app]. We have $e = \langle \lambda x.e'' \rangle\ w$ and $e' = e''[w/x]$. Moreover, from [e-app], we have $\sigma = \tau$ and $B \vdash \langle \lambda x.e'' \rangle : \tau' \rightarrow \tau$ and $B \vdash w : \tau'$. From [l-lam], we then have $B, x : \tau' \vdash e'' : \tau$. Now, from Proposition 2.5, we have $B \vdash e' : \sigma$, as required.

case [r-hc1]. We have $e = c\ v$ and $e' = c{\cdot}v$ and $ckind(B, c) = \mathtt{hub}$. Moreover, from [e-con1], we have $\sigma = \tau$ and $B(t)(c) \geq \tau' \rightarrow \tau$ and $\tau = \vec{\tau}\ t$ and $B \vdash v : \tau'$. It follows that we can apply [w-c1] to get $B \vdash e' : \sigma$, as required.

case [r-lcase0]. We have $e = \mathtt{case}\ {\cdot}c\ \mathtt{of}\ b_1\ \mid\ \dots\ \mid b_n$ and $ckind(B, c) = \mathtt{lub}$ and $b_i = c \Rightarrow e'$ and $1 \leq i \leq n$. Moreover, from [e-case], we have $\sigma = \tau$ and $\Gamma_i = (B, \vec{\tau}\ t) \triangleright c = \{\}$ and $B \vdash e' : \sigma$, as required.

The remaining cases follow similarly. □

Proposition A.1 (Value Boxity). *Assume* $B \vdash w : \tau$. *If* $B \vdash \tau : \mathtt{box}$ *then* w *is a boxed value. If* $B \vdash \tau : \mathtt{lub}$ *then* w *is a low-unboxed value (or a boxed value).*

Proof. By simple case analysis on the derivation of $B \vdash w : \tau$. □

Proposition 2.7 (Progress). *If* $\vdash B$ *and* $B \vdash e : \sigma$ *then either* e *is a value or there exists* e' *such that* $B \vdash e \rightsquigarrow e'$.

Proof. From Proposition 2.3, we have either e is a value of there exist a redex r, a type scheme σ', and a context E such that $B \vdash r : \sigma'$ and $e = E[r]$. We proceed by case analysis on the structure of the redex r. There are seven possibilities.

case $r = c\ w$. From [e-con1], we have there exists κ such that $B(t) = (\kappa, C)$ and $\sigma' = \tau'$ and $C(c) \geq \tau'' \rightarrow \tau'$ and $\tau' = \vec{\tau}\ t$ and $B \vdash w : \tau''$. There are now three sub-cases, one for each possible value of κ.

subcase $\kappa = \mathtt{box}$. In this case we can apply [r-bc1] to get there exists a boxed value $l = \{c, w\}$ such that $B \vdash r \rightsquigarrow l$. Now, from [r-ctx], we have $e' = E[l]$ and $B \vdash e \rightsquigarrow e'$, as required.

subcase $\kappa = \mathtt{lub}$. In this case, we need to ensure that w is a boxed value l. From $\vdash B$, it follows that $B \vdash t$ holds and because $C(c) \geq \tau'' \rightarrow \tau'$, it follows that rule [lub] must have been applied and, hence, that $B \vdash \tau'' : \mathtt{box}$ holds (type boxity is trivially closed under type substitution). We can now apply Proposition A.1 to get that w is a boxed value l. It then follows that we can apply [r-lc1] to get there exists a low-unboxed value $v = l{\cdot}c$ such that $B \vdash r \rightsquigarrow v$. Now, from [r-ctx], we have $e' = E[v]$ and $B \vdash e \rightsquigarrow e'$, as required.

subcase $\kappa = \mathtt{hub}$. In this case, we need to ensure that w is a low-unboxed value v. From $\vdash B$, it follows that $B \vdash t$ holds and because $C(c) \geq \tau'' \rightarrow \tau'$, it follows that rule [hub] must have been applied and, hence, that $B \vdash \tau'' : \mathtt{lub}$ holds (type boxity is trivially closed under type substitution). We can now apply Proposition A.1 to get that w is a low-unboxed value v. It then follows that we can apply [r-hc1] to get there exists a high-unboxed value $w' = c{\cdot}v$ such that $B \vdash r \rightsquigarrow w'$. Now, from [r-ctx], we have $e' = E[w']$ and $B \vdash e \rightsquigarrow e'$, as required.

case $r = \mathtt{let}\ x = w\ \mathtt{in}\ e_1$. From [r-let], we have $B \vdash r \rightsquigarrow e_1[w/x]$. Now, from [r-ctx], we have $e' = E[e_1[w/x]]$ and $B \vdash e \rightsquigarrow e'$, as required.

□

A.2 Properties and Proofs for Simulation

Proposition 2.8 (Simulation Value Substitution). *If $(B; e) \sim (B'; e')$ and $(B; w) \sim (B'; w')$ then $(B; e[w/x]) \sim (B'; e'[w'/x])$.*

Proof. By straightforward induction over the derivation $(B; e) \sim (B'; e')$. The only interesting case is the case for variables (rule [s-var]), in which case the conclusion $(B; e[w/x]) \sim (B'; e'[w'/x])$ follows directly from the assumption $(B; w) \sim (B'; w')$. □

Proposition 2.9 (Typing Preserved by Strong Simulation). *If $B, \Gamma \vdash e : \sigma$ and $(B; e) \approx (B'; e')$ then $B', \Gamma \vdash e' : \sigma$.*

Proof. By induction on the typing derivation for e. Each of the three representations of argument-carrying constructors results in three subcases (utilising the corresponding simulation rule and, perhaps, the symmetry rule [s-sym] for simulation), which are shown, straightforwardly, each using a single induction step. The only other interesting case is the case for variables, which is easily shown by the fact that simulation is reflexive in the variable case (rule [s-var]).[8] □

Proposition A.2 (Context Simulation). *If r is a redex and $(B, E[r]) \sim (B', e')$ then there exists a redex r' and a context E' such that $e' = E'[r']$ and $(B, r) \sim (B', r')$. Moreover, if $(B, E[r]) \sim (B', E'[r'])$ and $(B, r) \sim (B', r')$ and $(B, e) \sim (B', e')$ then $(B, E[e]) \sim (B', E'[e'])$.*

Proof. Each property is proven using straightforward induction over the structure of E. □

Proposition 2.10 (Evaluation Preserved by Strong Simulation). *If $B \vdash e : \sigma$ and $(B; e) \approx (B'; e')$ and $B \vdash e \rightsquigarrow e_2$ then there exists e'_2 st $B' \vdash e' \rightsquigarrow e'_2$ and $(B; e_2) \approx (B'; e'_2)$.*

Proof. From Proposition 2.3 and because e is clearly not a value, we have there exists a unique redex r, a type scheme σ_1, and a context E such that $B \vdash r : \sigma_1$ and $e = E[r]$. From Proposition A.2, we have that there exists a redex r' and a context E' such that $e' = E'[r']$ and $(B, r) \sim (B', r')$. By possible use of [r-ctx], we have $B \vdash r \rightsquigarrow e_1$ and $e_2 = E[e_1]$. We proceed by a case analysis of the possible instances of $(B, r) \sim (B', r')$.

case $r = \underline{c}$ and $r' = \underline{c}$. There are three subcases corresponding to the possible uses of [r-bc0], [r-lc0], and [r-hc0]. For all subcases, we have that there exists κ such that $ckind(B, c) = \kappa$. From the definition of similarity of boxity environments, we have (because $B \sim B'$) that there exists κ' such that $ckind(B', c) = \kappa'$. We proceed by case analysis on the different combinations of κ and κ'. There are a total of nine subcases.

subcase $\kappa = \mathsf{hub}$ and $\kappa' = \mathsf{box}$. From [r-hc0] and [r-bc0], we have $e_1 = c\cdot$ and there exists e'_1 such that $B' \vdash r' \rightsquigarrow e'_1$ and $e'_1 = \{c\}$. It now follows from [s-hb0] that $(B, e_1) \sim (B', e'_1)$. Moreover, from the second property of Proposition A.2, we can conclude that there exists $e'_2 = E'[e'_1]$ such that $(B, E[e_1]) \sim (B', e'_2)$ and hence $(B, E[e_1]) \approx (B', e'_2)$ because $B \sim B'$. We can also apply [r-ctx] to get $B' \vdash e' \rightsquigarrow e'_2$, as required.

subcase ... The remaining eight subcases proceed similarly (some subcases make use of rule [s-sym]).

case $r = \texttt{let}\ x = w\ \texttt{in}\ e_0$ and $r' = \texttt{let}\ x = w'\ \texttt{in}\ e'_0$. Only [s-let] applies. We have from [r-let] that $e_1 = e_0[w/x]$. We can also apply [r-let] to get that there exists $e'_1 = e'_0[w'/x]$ such that $B' \vdash r' \rightsquigarrow e'_1$. Moreover, from Proposition 2.8, we have $(B, e_1) \sim (B', e'_1)$. Now, from the second property of Proposition A.2, we can conclude that there exists $e'_2 = E'[e'_1]$ such that

[8] We assume that the symmetry rule is never applied more than once in sequence, which ensures that the simulation-relation is inductively well-founded. For simplicity, the rules provided do not include the extra tagging that, formally, can guarantee this property.

$(B, E[e_1]) \sim (B', e_2')$ and hence $(B, E[e_1]) \approx (B', e_2')$ because $B \sim B'$. We can also apply [R-CTX] to get $B' \vdash e' \rightsquigarrow e_2'$, as required.

The remaining cases follow similarly. □

A.3 Overview over Notation and Syntactic Constructs

This appendix provides an overview of notation and syntactic constructs defined in the paper.

$$
\begin{array}{lcll}
\tau & := & \alpha \mid \texttt{int} \mid \vec{\tau}\, t \mid \tau * \tau \mid \tau \to \tau & \text{types} \\
\sigma & := & \forall \vec{\alpha}.\tau & \text{type schemes} \\
\\
\kappa & ::= & \texttt{hub} \mid \texttt{lub} \mid \texttt{box} & \text{boxities} \\
\\
db & ::= & \vec{\alpha}\, t = C & \text{data bindings} \\
dd & ::= & \texttt{type}\ db_1\ \texttt{and}\ \ldots\ \texttt{and}\ db_n & \text{data declarations} \\
\\
l & ::= & \langle w, w\rangle \mid \langle \lambda x.e\rangle \mid \langle \texttt{fix}\ f\ x = e\rangle \mid \{c, w\} \mid \{c\} & \text{boxed values} \\
v & ::= & l \mid l{\cdot}c \mid {\cdot}c & \text{low-unboxed values} \\
w & ::= & v \mid c{\cdot}v \mid c{\cdot} \mid d & \text{high-unboxed values} \\
\\
e & ::= & x \mid w \mid c\ e \mid \underline{c} \mid e\ e & \text{expressions} \\
 & \mid & \texttt{let}\ x = e\ \texttt{in}\ e \mid (e, e) \mid \#i\ e & \\
 & \mid & \lambda x.e \mid \texttt{fix}\ f\ x = e & \\
 & \mid & \texttt{case}\ e\ \texttt{of}\ b \mid \ldots \mid b & \\
\\
b & ::= & p \Rightarrow e & \text{branches} \\
p & ::= & c\ x \mid c & \text{patterns} \\
P & ::= & dd\ P \mid e & \text{programs}
\end{array}
$$

α	:	type variable
$t^{(n)}$	:	type name of arity n
fc B	:	set of free value constructors in boxity environment B
ftv X	:	set of free type variables in object X
S	:	substitution mapping type variables to types
$\sigma \geq \tau'$	:	type τ' is an instance of type scheme σ
M, M'	:	map with $\mathrm{Dom}(M, M') = \mathrm{Dom}\, M \cup \mathrm{Dom}\, M'$ assuming $\mathrm{Dom}\, M \cap \mathrm{Dom}\, M' = \emptyset$ and values $(M, M')(x) = \begin{cases} M(x) & \text{if } x \in \mathrm{Dom}\, M \\ M'(x) & \text{otherwise} \end{cases}$
$ckind(B, c)$	=	κ if there exists t such that $B(t) = (\kappa, C)$ and $c \in \mathrm{Dom}\, C$
$(B, \vec{\tau}\, t) \triangleright p$	=	$\begin{cases} \{\} & \text{if } p = c \\ \{x : \tau\} & \text{if } p = c\ x \text{ and } B(t)(c) \geq \tau \to \vec{\tau}\, t \end{cases}$

References

Andrew W. Appel. 1992. *Compiling with continuations.* Cambridge University Press, USA.

Andrew W. Appel and David B. MacQueen. 1987. A Standard ML Compiler. In *Functional Programming Languages and Computer Architecture*, Gilles Kahn (Ed.). ACM, Springer-Verlag.

Thaïs Baudon, Laure Gonnord, and Gabriel Radanne. 2022. *Knit&Frog: Pattern matching compilation for custom memory representations.* Technical Report RR-9473. Inria Lyon. 35 pages. https://inria.hal.science/hal-03684334v3

Thaïs Baudon, Gabriel Radanne, and Laure Gonnord. 2023. Bit-Stealing Made Legal: Compilation for Custom Memory Representations of Algebraic Data Types. *Proc. ACM Program. Lang.* 7, ICFP, Article 216 (aug 2023), 34 pages. https://doi.org/10.1145/3607858

Lars Bergstrom and John Reppy. 2009. Arity raising in Manticore. In *Proceedings of the 21st International Conference on Implementation and Application of Functional Languages* (South Orange, NJ, USA) *(IFL'09)*. Springer-Verlag, Berlin, Heidelberg, 90–106.

Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. 1996. From Region Inference to von Neumann Machines via Region Representation Inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 171–183. https://doi.org/10.1145/237721.237771

Alex Bradbury. 2023. Storing data in pointers. https://muxup.com/2023q4/storing-data-in-pointers Posted on Muxup: Adventures in collaborative open source development (2023Q4).

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (sep 2009), 509–543. https://doi.org/10.1017/S0956796809007205

Nicolas Chataing, Stephen Dolan, Gabriel Scherer, and Jeremy Yallop. 2024. Unboxed Data Constructors: Or, How cpp Decides a Halting Problem. *Proc. ACM Program. Lang.* 8, POPL, Article 51 (jan 2024), 31 pages. https://doi.org/10.1145/3632893

Simon Colin, Rodolphe Lepigre, and Gabriel Scherer. 2019. Unboxing Mutually Recursive Type Definitions in OCaml. In *JFLA 2019 - 30 èmes journées francophones des langages applicatifs*. Les Rousses, France. https://inria.hal.science/hal-01929508

Karl Crary. 2003. Toward a foundational typed assembly language. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. Association for Computing Machinery, New York, NY, USA, 198–212. https://doi.org/10.1145/604131.604149

Paul Downen, Zena M. Ariola, Simon Peyton Jones, and Richard A. Eisenberg. 2020. Kinds are calling conventions. *Proc. ACM Program. Lang.* 4, ICFP, Article 104 (aug 2020), 29 pages. https://doi.org/10.1145/3408986

Richard A. Eisenberg, Stephen Dolan, and Leo White. 2022. Unboxed Types for OCaml. In *Proceedings of the ML Family Workshop* (Ljubljana, Slovenia) *(ML '22)*.

Richard A. Eisenberg and Simon Peyton Jones. 2017. Levity polymorphism. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 525–539. https://doi.org/10.1145/3062341.3062357

Martin Elsman. 1998. Polymorphic Equality - No Tags Required. In *Proceedings of the Second International Workshop on Types in Compilation (TIC '98)*. Springer-Verlag, Berlin, Heidelberg, 136–155.

Martin Elsman. 1999. Static interpretation of modules. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming* (Paris, France) *(ICFP '99)*. Association for Computing Machinery, New York, NY, USA, 208–219. https://doi.org/10.1145/317636.317800

Martin Elsman. 2008. *A Framework for Cut-Off Incremental Recompilation and Inter-Module Optimization.* Technical Report. IT University of Copenhagen, Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark.

Martin Elsman. 2024. Artifact for the ICFP 2024 paper: Double-Ended Bit-Stealing for Algebraic Data Types. Zenodo. https://doi.org/10.5281/zenodo.12684335

Martin Elsman and Niels Hallenberg. 2021. Integrating region memory management and tag-free generational garbage collection. *Journal of Functional Programming* 31 (2021), e4. https://doi.org/10.1017/S0956796821000010

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.* 18, 2 (mar 1996), 109–138. https://doi.org/10.1145/227699.227700

Niels Hallenberg, Martin Elsman, and Mads Tofte. 2002. Combining Region Inference and Garbage Collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 141–152. https://doi.org/10.1145/512529.512547

Robert Harper and Greg Morrisett. 1995. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 130–141. https://doi.org/10.1145/199448.199475

Fritz Henglein and Jesper Jørgensen. 1994. Formally optimal boxing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 213–226. https://doi.org/10.1145/174675.177874

Intel. 2024. Intel 64 and IA-32 Architectures Software Developer's Manual. https://cdrdv2.intel.com/v1/dl/getContent/671200 Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4.

Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC '02)*. USENIX Association, USA, 275–288.

Mark P. Jones. 1994. A theory of qualified types. *Science of Computer Programming* 22, 3 (1994), 231–256. https://doi.org/10.1016/0167-6423(94)00005-0

Chaitanya Koparkar, Mike Rainey, Michael Vollmer, Milind Kulkarni, and Ryan R. Newton. 2021. Efficient tree-traversals: reconciling parallelism and dense data representations. *Proc. ACM Program. Lang.* 5, ICFP, Article 91 (aug 2021), 29 pages. https://doi.org/10.1145/3473596

Xavier Leroy. 1990. *The ZINC experiment : an economical implementation of the ML language*. Technical Report RT-0117. INRIA. 100 pages. https://inria.hal.science/inria-00070049

Xavier Leroy. 1992. Unboxed Objects and Polymorphic Typing. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 177–188.

Xavier Leroy. 1997. The effectiveness of type-based unboxing. In *Proceedings of the 1997 ACM Workshop on Types in Compilation* (Amsterdam). https://api.semanticscholar.org/CorpusID:2398412

Linux Kernel Development Community. 2024. The Linux Kernel documentation. https://docs.kernel.org/arch/x86/x86_64/mm.html Section 29.3.1 on Memory Management.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML (Revised)*. MIT Press.

Greg Morrisett. 1995. *Compiling with Types*. Ph. D. Dissertation. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

Donald R. Morrison. 1968. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 4 (oct 1968), 514–534. https://doi.org/10.1145/321479.321481

Huu-Duc Nguyen and Atsushi Ohori. 2006. Compiling ML polymorphism with explicit layout bitmap. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Venice, Italy) *(PPDP '06)*. Association for Computing Machinery, New York, NY, USA, 237–248. https://doi.org/10.1145/1140335.1140364

Chris Okasaki. 1999. *Purely Functional Data Structures*. Cambridge University Press, USA.

Chris Okasaki and Andy Gill. 1998. Fast mergeable integer maps. In *ACM SIGPLAN Workshop on ML*. 77–86.

John Peterson and Mark Jones. 1993. Implementing Type Classes. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 227–236. https://doi.org/10.1145/155090.155112

Simon L. Peyton Jones and John Launchbury. 1991. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture* (Cambridge, Massachusetts, USA). Springer-Verlag, Berlin, Heidelberg, 636–666.

Rust Community. 2021. RFC: Alignment niches for references types. https://github.com/rust-lang/rfcs/pull/3204 Rust RFC.

David R. Tarditi and Andrew W. Appel. 2000. ML-Yacc User's Manual. Microsoft Research and Department of Computer Science, Princeton University.

Peter J. Thiemann. 1995. Unboxed values and polymorphic typing revisited. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, USA) *(FPCA '95)*. Association for Computing Machinery, New York, NY, USA, 24–35. https://doi.org/10.1145/224164.224175

Mads Tofte and Lars Birkedal. 1998. A Region Inference Algorithm. *ACM Trans. Program. Lang. Syst.* 20, 4 (jul 1998), 724–767. https://doi.org/10.1145/291891.291894

Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. 2022. *Programming with Regions in the MLKit (Revised for Version 4.7.2)*. Technical Report. IT University of Copenhagen, Denmark.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (1997), 109–176. https://doi.org/10.1006/inco.1996.2613

Stephen Weeks. 2006. Whole-Program Compilation in MLton. In *Proceedings of the 2006 Workshop on ML* (Portland, Oregon, USA) *(ML '06)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/1159876.1159877

A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (nov 1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Lukasz Ziarek, Stephen Weeks, and Suresh Jagannathan. 2008. Flattening tuples in an SSA intermediate representation. *Higher Order Symbol. Comput.* 21, 3 (sep 2008), 333–358. https://doi.org/10.1007/s10990-008-9035-3

Received 2024-02-28; accepted 2024-06-18