Unboxed Polymorphic Objects for Functional Numerical Programming

Christopher Rodrigues
cirodrig@illinois.edu

Wen-Mei Hwu
w-hwu@illinois.edu

IMPACT Technical Report
IMPACT-12-02
University of Illinois at Urbana-Champaign

August 24, 2012

Abstract

One approach to making numerical algorithms easier to write and understand is to provide a high-level library of polymorphic container types and higher-order functions for manipulating collections of data. Although polymorphic functions and data types are a common feature of programming languages, programming language implementations conventionally rely on compile-time knowledge of data layout when generating machine code. Ensuring that sizes are known at compile time requires boxing objects, which sacrifices performance; dynamically compiling functions, which requires heavyweight runtime support; prohibiting first-class polymorphism, which limits expressiveness; or using a predetermined set of monomorphic data types, which limits the range of usable types.

To use efficient data layouts in statically compiled polymorphic code without limiting the use of polymorphism, we propose a low-level language that allows object sizes to depend on run-time type information. This language is used as an intermediate language in our prototype compiler, Triolet. So that users do not need to be aware of the details of data layout and object sizes, we further propose a translation from a high-level, fully boxed language to the internal language. To evaluate Triolet, we measured the performance of a number of functional numerical algorithms. These algorithms use a library of data types and functions, some of which utilize first-class polymorphism. These algorithms achieve 52% of the performance of equivalent monomorphic implementations in C, often allocating the same amount of memory. In contrast, using boxed arrays slows down Triolet by 2.87×. These results demonstrate that it is possible to achieve efficient data layout in this style of numerical code, and good data layout is key to achieving good performance.

1 Introduction

Functional programming makes it possible to write numerical codes in a clear, simple, declarative style. It's possible to specify algorithms in terms of collective operations—maps, reductions, and so forth—that transform entire arrays from one meaningful form to another, instead of using low-level array reads and writes to incrementally alter arrays. This has motivated the development of numerous numerical functional programming languages and libraries [15, 2, 34, 7, 20]. Collective operations are typically polymorphic so that they can be reused on arbitrary data types. If collective operations are to be the building blocks of numerical algorithms that are as fast as what can be produced

in a low-level, imperative programming language such as C, polymorphism must not impose a significant run-time cost. In particular, the underlying language must support efficient representations of polymorphic data structures.

Unfortunately, the go-to method of implementing polymorphism is to *box* objects whose types are not statically known. A boxed object is stored in a chunk of heap-allocated memory and referenced by a pointer. Boxed objects are costly to use: each boxed object has to be allocated and later garbage collected, and an extra pointer indirection is needed to access its data. Parametrically polymorphic container types such as Java's `ArrayList` and Haskell's `Array` keep their contents in boxed form. In applications that chiefly operate on arrays of numbers, the overhead of boxing and unboxing individual array elements can dominate a program's execution time [28, 43, 5].

Previous work avoids boxing by generating unboxed *monomorphic* code where the original program is polymorphic. Generating monomorphic code requires prohibiting first-class polymorphism [39, 15, 2], deferring compilation until run time [21, 33], or selecting from a limited set of predefined unboxed interfaces using compile-time specialization [14], overloaded interfaces [20], or run-time inspection of tags [27, 25] or types [18]. These approaches all have drawbacks: they limit a language's expressiveness, they rely on a heavyweight just-in-time compilation infrastructure, or they can only unbox a small subset of the types available to users.

We take a more direct approach by supporting unboxed polymorphism natively in our programming language, Triolet. Triolet is a subset of Haskell with strict semantics and a syntax inspired by Python. The efficient representation of data structures brings the performance of functional programming with collective operations nearly on par with imperative loops written in C. Numerical algorithms written in the Triolet language are statically compiled to code that runs at 52% (geomean) the speed of the same algorithms in C. Moreover, polymorphic unboxing provides a necessary performance boost: those Triolet algorithms would run 2.87× slower if all arrays were boxed. Both user code and the library of collective operations are compiled by the Triolet compiler. While the generated code is mostly monomorphic, support for first-class polymorphism is required due to the use of type classes [42] and iterators [10] in the library interface.

The challenge of implementing the combination of polymorphism and unboxing is that an object's memory layout may not be statically known. Consequently, a compiler cannot statically compute the sizes of objects or the positions of an object's fields, which are needed in order to generate address computation and memory operations. Triolet treats an unboxed object as a chunk of bytes with a size determined by its type and embeds run-time size computation into programs in a manner similar to type-passing [18]. Primitive operations for creating and accessing objects take sizes as extra run-time parameters. The only data types that cannot be unboxed are recursive data types and those whose type does not carry information about their memory layout (e.g., functions).

Immutable objects, in programming languages that have them, must be created and initialized in a single step so that a program only sees fully built objects. This is typically done by copying fields into an object when it is created, but in an unboxed setting this would lead to unnecessary object creation and copying. Triolet allows objects to be constructed inside other objects, rather than copied, so that unboxed fields can be written efficiently without violating immutability.

To represent unboxed object accesses internally, the Triolet compiler uses a variant of System F^ω that we call System F^ω_U, for "System F^ω with unboxing". System F^ω_U supports boxed and unboxed types, higher-kinded types, arrays, algebraic data types, existential types, and first-class polymorphism in a type-safe way. Since the compiler's internal representation is close to other System F^ω-based languages, it can employ existing, general-purpose optimization techniques [40, 30].

Users do not see any additional complexity when writing functions in Triolet. Triolet programmers write as if all objects were boxed. Algebraic data types are defined in System F_U^ω and can be used in Triolet code. The compiler translates Triolet to System F_U^ω, unboxing object fields whenever possible. More precisely, an object is unboxed if it has an unboxed type and is put into a field that has an unboxed type. The compiler inserts run-time size computation, selects boxed and unboxed representations, and coerces [24] between representations as needed.

The implementation of System F_U^ω is ongoing. Our experimental results are generated using a prototype Triolet compiler. As we describe in Section 8, the prototype compiles Triolet code using System F_U^ω as an intermediate language, but it has some limitations with regard to analyzing data type definitions and compiling code written directly in F_U^ω.

In summary, our contributions are the following.

- We introduce System F_U^ω, the first programming language that supports polymorphism over unboxed types without relying on a translation to monomorphic code.
- We present an algorithm to compile System F_U^ω's primitive data structure operations to memory loads, stores, and address calculation.
- We present a translation algorithm from a fully boxed language to System F_U^ω, allowing users to take advantage of unboxed polymorphism without managing unboxing explicitly.
- We demonstrate that functional numerical algorithms written using polymorphic libraries in Triolet achieve 52% the performance of handwritten C code.

2 Preliminaries

Some notational conventions are common to the several intermediate languages that we discuss. Type parameters and type application are explicit. For variables, we use letters like x or sans-serif words with a lowercase initial like $\mathsf{repTuple}$. Type variables are Greek letters like α. Type constructors are sans-serif words with an uppercase initial like Tuple. Data constructors are sans-serif words with a lowercase initial like tuple. When a type constructor has a single data constructor associated with it, we give the same name (modulo capitalization) to both. Function application on the left-hand side of an equation is shorthand for lambda abstraction, e.g.,

$\mathsf{id} : \forall \alpha : \star.\ \alpha \to \alpha$
$\mathsf{id}\ \alpha\ x = x$

is shorthand for $\mathbf{let}\ \mathsf{id} = \Lambda \alpha : \star.\ \lambda x : \alpha.\ x$.

We use substitutions to instantiate polymorphic types. Given a substitution θ, we write $\theta(\tau)$ for the type produced by applying the substitution θ to τ. We write $[\pi/\alpha]$ for the substitution that replaces each occurrence of α by π, and $[\overline{\pi/\alpha}]$ for the substitution that replaces each occurrence of an α from the list $\overline{\alpha}$ by the corresponding type in $\overline{\pi}$.

Some algorithms are type-directed, meaning that type inference, kind inference, or a similar task is run as part of the algorithm. When presenting an algorithm, we put subtasks in a "where" clause after an equation. To illustrate, the definition

$$\mathrm{foo}(\Gamma, \tau) = \mathrm{bar}(\kappa, \tau)$$
$$\text{where } \Gamma \vdash \tau : \kappa$$

should be read as defining an algorithm, foo, that runs kind inference on the type τ to infer its kind κ, which is passed to bar.

3 The Role of Polymorphism in Functional Numerical Programming

Triolet encourages a programming style that heavily utilizes polymorphic functions and admits no guaranteed translation to monomorphic code. Consequently, the performance of polymorphic operations has a large impact on Triolet's overall performance, which makes the support for unboxed polymorphism important. We briefly outline the role of polymorphism in Triolet programming.

Functional programming allows the clear and concise expression of programs by decomposing them into modular parts [19]. Some styles of numerical functional programming employ higher-order functions that operate on collections of data [37]. Such collective operations capture common computational structures while abstracting over the computation performed within those structures. For example, consider the task of finding the shortest nonempty array in a given array s of arbitrary-length arrays. We'll decompose this task into simpler parts, using iterators to pass sequences of data from one task to the next and collective operations to build transformations out of operations on individual pieces of data.

To get the nonempty arrays from the input, we use the collective operation filter. Given a predicate f and iterator x, filter f x returns an iterator over only those members of x that satisfy the predicate. That is, a member y appears in the output if and only if f y returns true. An array a is nonempty if its length is nonzero:

$$\mathsf{nonempty}\ a = \mathsf{length}\ a \neq 0$$

Putting the parts together, filter nonempty x returns the nonempty arrays from iterator x.

To get the shortest array from an iterator, we use an associative reduction to compare pairs of arrays repeatedly, keeping the shorter each time, until all arrays have been examined. The shortest of two arrays is the one with a smaller length:

$$\mathsf{shortest}\ a\ b = \mathbf{if}\ \mathsf{length}\ a \leqslant \mathsf{length}\ b\ \mathbf{then}\ a\ \mathbf{else}\ b$$

The collective operation reduce1 uses a combining function to reduce an iterator down to a single value. The shortest array in an iterator x is computed by reduce1 shortest x. It is more typical to define reduction to take a third argument, the combining function's identity value. This two-argument variant is useful with combining functions like shortest that have no identity value.

We assemble these parts into a function that iterates over its input (by calling iter), extracts the nonempty elements, and finds the shortest, shown below.

$$\begin{aligned}\mathsf{shortestArray}\ s = {} & \mathbf{let}\ \mathsf{nonempty}\ a = \mathsf{length}\ a \neq 0\ \mathbf{in} \\ & \mathbf{let}\ \mathsf{shortest}\ a\ b = \mathbf{if}\ \mathsf{length}\ a \leqslant \mathsf{length}\ b\ \mathbf{then}\ a\ \mathbf{else}\ b\ \mathbf{in} \\ & \mathsf{reduce1}\ \mathsf{shortest}\ (\mathsf{filter}\ \mathsf{nonempty}\ (\mathsf{iter}\ s))\end{aligned}$$

Collective operations are polymorphic; they can manipulate arbitrary data types because they stipulate nothing about what type of data a collection may contain. When filtering, filter does not directly inspect values, but rather calls its predicate argument to extract information from a value. When reducing, reduce1 selects values to combine, but calls its combining function to actually create a combined value. Typically, a language implementation would box values in polymorphic code so that filter and reduce1 can pass pointers around. Similarly, iter would expect an array of pointers as its argument. By supporting polymorphic unboxing, Triolet allows iter to access unboxed arrays. Temporary values are initially boxed—conventional optimizations can unbox them.

In a suitably limited language, it is possible for a compiler to eliminate polymorphism by replicating code for each data type, called *monomorphization*. However, monomorphization is not possible in languages with polymorphic values [26] because a program may create types on the basis of run-time decisions.

Iterators, used by Triolet's library interface, are one data type that cannot be monomorphized. As in prior work [10], iterators are polymorphic because they have hidden state in the form of existentially typed fields. From a programmer's point of view, all iterators that produce the same data type can be used interchangeably. For instance, given some iterators i and j, we could write **if** b **then** i **else** j to select one of the two iterators depending on the run-time value of b. When a collective operation runs an iterator, it keeps track of the iterator's state; but since **if** b **then** i **else** j may have one of two different state types, the state type is unknown to the collective operation and must be boxed. Optimizations can sometimes identify an existentially bound type and use that knowledge to generate unboxed monomorphic code, but this is not always possible.

Triolet supports operator overloading via type classes, which require support for first-class polymorphic functions [29]. An overloaded function call translates to code that dynamically looks up a function, then calls it. For example, the iter function is overloaded to work with various array-like container types. When constructing an iterator over an array, the run-time lookup returns the internal library function iterArray; when constructing an iterator over an iterator, it returns iterIterator. Each of these library functions is polymorphic, able to use an array (or iterator) of any data type. Thus, iterating over an array utilizes a first-class polymorphic method.

In summary, the programming abstractions that we wish to support require first-class polymorphism. Optimizations can convert from polymorphic to monomorphic code, but it is impossible to do so in all cases. Consequently, we support polymorphism throughout the Triolet compiler. We wish to generate working code for all valid inputs, and to generate fast code in the common case where the compiler can eliminate polymorphism.

4 Overview of Unboxed Polymorphism in the Triolet Compiler

The discussion of how Triolet supports unboxed polymorphism is organized into three sections.

Section 5 discusses support for unboxed polymorphism in the context of a functional language. We point out some aspects of polymorphism that are easy in a fully boxed language but become tricky when unboxing is allowed. For concreteness, data structure access is discussed in terms of λ^{L}, a simplified form of Triolet's backend language, which accesses memory through explicit pointer arithmetic, loads, and stores. We then introduce a model of memory layout that bridges the notions of memory operations and immutable data structures and makes these tricky aspects of polymorphism easier to reason about.

Section 6 presents the compiler's internal language, System F_U^ω. The first few subsections describe the kind system, the way objects are laid out in memory, and the type system. The type system is unusual in its dependence on the low-level representation of data types (Section 6.4). Before type checking, data type definitions are analyzed to produce constructor signatures (Section 6.4.3), which are used in the typing judgments for operations on data types (Section 6.5). Data type definitions are also analyzed to determine how operations on data types are lowered to explicit memory operations. The algorithm for lowering code from System F_U^ω to λ^{L} is presented in Section 6.6.

Section 7 presents the translation from Triolet to System F_U^ω. The translation is

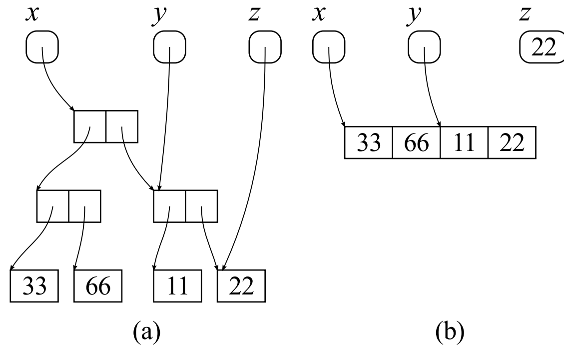

Figure 1: An array of tuples of floats in different storage strategies. Rounded rectangles are variables. Other rectangles are heap data. In part (a), all objects are boxed. In part (b), tuples are bare and floats are values.

guided by analysis of data type definitions (Section 7.2) and by the type signatures of library functions written in System F^{ω}_{U}. The first step of the translation is to insert run-time type information into a Hindley-Milner typeable [12] input language. This is implemented as an extension to type inference (Section 7.3). The second step is to identify data in the fully boxed code that should be unboxed and represent it in unboxed form (Section 7.4). This step introduces the code necessary to manipulate unboxed types, and to use boxed and unboxed types together.

5 Unboxed Polymorphism

A functional programming language provides the illusion that a variable can contain an arbitrarily complex piece of data. In reality, data is decomposed into discrete pieces linked by pointers. The choice of where to place data and where to use pointers has consequences for the performance and complexity of a language implementation. In Triolet, there are three ways that storage may be reserved for data, or *storage strategies*. A *boxed* object is represented as a heap-allocated chunk of memory. We say that a language is fully boxed if the language boxes all objects. A *bare* object is stored directly in another object in memory. Bare objects reside inside boxed objects (possibly nested in other bare objects). Variables point to, but never contain, boxed or bare objects. A *value* object is stored directly in another object or in a variable. Since Triolet variables are register allocated in the final stages of compilation, value variables are the closest thing to register storage that we consider.

Figure 1 illustrates how the array of tuples $[(33, 66), (11, 22)]$ can be represented in different ways by using different storage strategies for its components. Part (a) illustrates a memory organization where every object is boxed. The array, tuples, and integers are individually allocated. Part (b) illustrates a memory organization where the array is boxed, the tuples are bare, and the integers are values. There is only one block of heap-allocated memory, and integers occupy registers for instant access.

We use the low-level imperative language λ^{L}, shown in Figure 2, to demonstrate how polymorphism may be handled using pointer arithmetic, loads, and stores. λ^{L} is monomorphic and does not associate type information with pointers or memory locations. Polymorphism is implemented by manipulating pointers that point to unknown data. Variables hold data that can be put in registers: integers, pointers, the unit value, and products of these. Functions have pointer type. Products are unpacked using the multi-assignment pattern **let** $x \times y = \ldots$ **in** $\ldots$. In types, values, and patterns, we abbreviate

Types	$\sigma := \mathsf{Int} \mid \mathsf{Ptr} \mid \langle\rangle \mid \sigma \times \sigma$
Values	$v := x \mid N \mid \langle\rangle \mid v \times v$
Patterns	$p := v \mid \langle\rangle \mid p \times p$
Expressions	$e := v \mid e\ e \mid e\ op\ e \mid \mathbf{load}_\sigma\ e \mid \mathbf{store}_\sigma\ e\ e \mid \mathbf{alloc}\ e$
	$\mid \lambda x : \sigma.\ e \mid \mathbf{let}\ p = e\ \mathbf{in}\ e$

Figure 2: Syntax of λ^{L}, a low-level, load-store language.

nested products $\sigma_1 \times (\sigma_2 \times \cdots)$ as $\langle \sigma_1, \sigma_2, \ldots \rangle$ or $\langle \overline{\sigma} \rangle$. There are the usual primitive operations for integer and pointer arithmetic. For looping, we assume a predefined function loop such that loop $f\ n$ performs the sequence of calls $(f\ 0), (f\ 1), \ldots, (f\ (n-1))$ and returns $\langle\rangle$.

We make some simplifying assumptions in the presentation. Whereas λ^{L} permits arbitrary nesting of expressions and lambda functions, the Triolet compiler maintains expressions in ANF [16] as its more rigid structure simplifies compilation to C. Conditional execution and recursive function definitions are omitted from λ^{L}; they present no additional challenge to supporting polymorphism. Pointers, integers, and the unit type are the only primitive types in λ^{L}. Memory is assumed to be word-addressed with pointers and integers occupying one word. Support for primitive types with different sizes and alignments is orthogonal to the topic of this paper.

5.1 Accommodating Polymorphism

When we don't have the luxury of representing everything as a boxed object, implementing polymorphism becomes tricky. Some objects are boxed and represented by a pointer, while others are unboxed and represented in-place; polymorphic code has to deal with both. Getting polymorphic code to work with unboxed data boils down to two issues. First, object sizes must be passed around explicitly in order to do address computation. Second, parameter passing must be handled consistently.

In a fully boxed language, we could define the following function that uses a function f to process the contents of a tuple. The function is polymorphic; parameters α and β give the types of the tuple fields, and γ gives the return type. The case expression reads the fields of tuple t, binding each one to a variable.

$$\mathsf{tupleApply} : \forall \alpha, \beta, \gamma : \star.\ (\alpha \to \beta \to \gamma) \to \mathsf{Tuple}\ \alpha\ \beta \to \gamma$$
$$\mathsf{tupleApply}\ \alpha\ \beta\ \gamma\ f\ t = \mathbf{case}\ t\ \mathbf{of}\ \mathsf{tuple}\ x\ y.\ f\ x\ y$$

When data structure accesses are translated into explicit pointer operations, the following λ^{L} code is produced. Since object fields are boxed, x and y are each pointers to boxed objects. These pointers are loaded and passed as arguments to f.

$$\begin{aligned}
&\mathsf{tupleApply} : \mathsf{Ptr} \to \mathsf{Ptr} \to \mathsf{Ptr} \\
&\mathsf{tupleApply}\ f\ t = \mathbf{let}\ x = \mathbf{load}_{\mathsf{Ptr}}\ t\ \mathbf{in} \\
&\qquad\qquad\qquad\quad \mathbf{let}\ y = \mathbf{load}_{\mathsf{Ptr}}\ (t+1)\ \mathbf{in} \\
&\qquad\qquad\qquad\quad f\ x\ y
\end{aligned}$$

To support tuples of bare objects, we have to adjust both the definition of tupleApply and the definition of whatever functions are passed as f. A polymorphic tuple consists of two objects laid out consecutively in memory. Because the object contents are unknown, polymorphic code can only compute their addesses and pass them along. The object

sizes s_1 and s_2 are passed as extra parameters to tupleApply. (Both sizes are passed, though only the first is used in this case.) The parameters to f are derived from the sizes and the pointer t using pointer arithmetic.

$\mathsf{tupleApply} : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Ptr} \to \mathsf{Ptr} \to \mathsf{Ptr}$
$\mathsf{tupleApply}\ s_1\ s_2\ f\ t = f\ t\ (t + s_1)$

Responsiblity for loading the fields is offloaded to f, so we now look at two functions that could be passed as f. To add a tuple of *unboxed* integers, we would pass the following function as f. It loads the values, adds them, puts the result in a newly allocated object, and returns a pointer to it. These memory operations convert between storage strategies: the inputs are bare (held in fields of an object), addition operates on values (held directly in variables), and the return value is boxed (allocated on the heap). Unlike in tupleApply, there are no size parameters because sizes are known statically. The return type is known to be Int, so its size appears as the constant 1.

$$
\begin{aligned}
&\mathsf{my_add} : \mathsf{Ptr} \to \mathsf{Ptr} \to \mathsf{Ptr}\\
&\mathsf{my_add}\ p\ q = \mathbf{let}\ m = \mathbf{load}_{\mathsf{Int}}\ p\ \mathbf{in}\\
&\qquad\qquad\quad \mathbf{let}\ n = \mathbf{load}_{\mathsf{Int}}\ q\ \mathbf{in}\\
&\qquad\qquad\quad \mathbf{let}\ r = \mathbf{alloc}\ 1\ \mathbf{in}\\
&\qquad\qquad\quad \mathbf{let}\ \langle\rangle = \mathbf{store}_{\mathsf{Int}}\ r\ (m + n)\ \mathbf{in}\\
&\qquad\qquad\quad r
\end{aligned}
$$

To compute the composition of two *boxed* functions stored in a tuple, we would start with a tuple holding two pointers to functions (recall that boxed fields are represented as pointers). The function argument of tupleApply receives pointers to these fields. Given two pointers g and h to functions that operate on boxed objects, their composition is $\lambda x{:}\mathsf{Ptr}.g\ (h\ x)$. However, tupleApply does not pass pointers to functions as the arguments of f; it passes pointers to pointers to functions. To match up calling conventions, we can build a function my_compose that loads from its arguments and then composes the two functions. This function can be passed as the parameter to tupleApply.

$$
\begin{aligned}
&\mathsf{my_compose} : \mathsf{Ptr} \to \mathsf{Ptr} \to \mathsf{Ptr}\\
&\mathsf{my_compose}\ p\ q = \mathbf{let}\ g = \mathbf{load}_{\mathsf{Ptr}}\ p\ \mathbf{in}\\
&\qquad\qquad\qquad\ \ \mathbf{let}\ h = \mathbf{load}_{\mathsf{Ptr}}\ q\ \mathbf{in}\\
&\qquad\qquad\qquad\ \ \lambda x : \mathsf{Ptr}.\ g\ (h\ x)
\end{aligned}
$$

As my_concatenate and my_add show, storage strategy conversions are sometimes required when interfacing with polymorphic code.

5.2 Initializing Objects Efficiently

Initializing bare objects in a language with immutable data, such as Triolet, is a delicate matter. Triolet objects are created and initialized in a single step, and cannot be modified thereafter. Objects are created by primitive operations (for algebraic data types) or library functions defined in λ^{L} (for other data types). In the usual formulation of algebraic data types [22], an object-constructing operation takes field values as arguments and returns a newly constructed object containing those values. For instance, in a fully boxed language, the following expression constructs the tuple $(1, (2, 3))$.

$\mathsf{tuple\ Int\ (Tuple\ Int\ Int)}\ 1\ (\mathsf{tuple\ Int\ Int}\ 2\ 3)$

The tuple data constructor constructs a tuple, given the types and values of its fields. First, the values 2 and 3 are assembled into $(2, 3)$. Then, 1 and $(2, 3)$ are assembled into the final tuple.

One cannot construct bare tuples in this order. The inner tuple is to be created first; however, it will be stored in the outer tuple, which is not created yet. A workaround would be to allocate temporary storage for the inner tuple, then copy it into the outer tuple. However, if the Triolet compiler were to allocate temporary storage for each bare object, it would perform as many allocations as a fully boxed language!

A better solution is to start by allocating storage for the outermost object, then write the integers 1, 2, and 3 at the right offsets in that object. λ^{L} can certainly do this. However, in System F_U^ω these operations must be expressed in terms of immutable objects, because the correctness of optimizations depends on objects being immutable. We change the semantics of data-constructing operations to initialize bare objects in place without exposing side effects in System F_U^ω. New bare objects are encoded as *initializers*. An initializer is a one-parameter function that takes a pointer to some uninitialized memory, and writes a bare object into it. Converting the fully-boxed expression above to use bare objects and translating it to λ^{L} yields the following code.

$$
\begin{aligned}
&\mathbf{let}\ t = (\lambda r : \mathsf{Ptr}.\ \mathbf{let}\ \langle\rangle = \mathbf{store}_{\mathsf{Int}}\ r\ 2\ \mathbf{in} \\
&\qquad\qquad\qquad \mathbf{store}_{\mathsf{Int}}\ (r+1)\ 3)\ \mathbf{in} \\
&\lambda s : \mathsf{Ptr}.\ \mathbf{let}\ \langle\rangle = \mathbf{store}_{\mathsf{Int}}\ s\ 1\ \mathbf{in} \\
&\qquad t\ (s+1)
\end{aligned}
$$

The two tuples have been translated to functions. The first one, when passed a pointer r, constructs a tuple at r by writing 2 to its first field and 3 to its second. The second one, when passed a pointer s, writes 1 to its first field and calls the first initializer to write its second field. Triolet's backend inlines the application $t\ (s+1)$ to produce the following function.

$$
\begin{aligned}
&\lambda s : \mathsf{Ptr}.\ \mathbf{let}\ \langle\rangle = \mathbf{store}_{\mathsf{Int}}\ s\ 1\ \mathbf{in} \\
&\qquad \mathbf{let}\ \langle\rangle = \mathbf{store}_{\mathsf{Int}}\ (s+1)\ 2\ \mathbf{in} \\
&\qquad \mathbf{store}_{\mathsf{Int}}\ (s+2)\ 3
\end{aligned}
$$

This function creates the tuple by writing three integers to memory. By constructing initializers instead of bare objects, the Triolet compiler can generate code for each tuple individually, while still writing data directly into complex objects.

An object can be created from the initializer by allocating memory and passing the address to the initializer. Abbreviating the expression above as X, the tuple can be created by

$$
\begin{aligned}
&\mathbf{let}\ p = \mathbf{alloc}\ 3\ \mathbf{in} \\
&\mathbf{let}\ \langle\rangle = X\ p\ \mathbf{in} \\
&p.
\end{aligned}
$$

An initializer can also be created from a bare object. This is done to copy an object. The function below copies from p to r. Applying it to one parameter produces an initializer that makes a copy of that parameter.

$$
\begin{aligned}
&\lambda p : \mathsf{Ptr}.\ \lambda r : \mathsf{Ptr}.\ \mathbf{let}\ \langle\rangle = \mathbf{store}_{\mathsf{Int}}\ r\ (\mathbf{load}_{\mathsf{Int}}\ p)\ \mathbf{in} \\
&\qquad\qquad\qquad \mathbf{let}\ \langle\rangle = \mathbf{store}_{\mathsf{Int}}\ (r+1)\ (\mathbf{load}_{\mathsf{Int}}\ (p+1))\ \mathbf{in} \\
&\qquad\qquad\qquad \mathbf{store}_{\mathsf{Int}}\ (r+2)\ (\mathbf{load}_{\mathsf{Int}}\ (p+2))
\end{aligned}
$$

5.3 Unboxed Arrays

While Triolet's unboxed arrays are not algebraic data types, they work similarly. System F_U^ω and Triolet code accesses arrays through library functions that are defined in λ^{L}. The library functions utilize dynamic size information for address computation and

initializers to write results directly into arrays. As with tuples, array construction and array reading does not allocate memory.

Reading an unboxed array performs address arithmetic on the array pointer to get a pointer to an array element. The function takes the array element size, an array index, and an array pointer, and returns an element pointer.

$\mathsf{index} : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Ptr} \to \mathsf{Ptr}$
$\mathsf{index}\ s\ i\ p = p + s * i$

Arrays can be constructed in several ways; a common one is to tabulate a function. Tabulation takes an array element size s, an array size n, and a function f. At each index i in a new array r, it writes the value returned by $f\ i$.

$\mathsf{tabulate} : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Ptr} \to \mathsf{Ptr} \to \langle\rangle$
$\mathsf{tabulate}\ s\ n\ f\ r = \mathsf{loop}\ n\ (\lambda i : \mathsf{Int}.\ f\ i\ (r + s * i))$

In the loop body, $f\ i$ is called to compute an initializer. This initializer is passed the address of an array element, $r + s * i$. The tabulate function itself returns an initializer when partially applied, since it creates an array at the address that r points to.

5.4 From Loads and Stores to Data Types

When explaining how to access tuples and arrays in λ^{L}, we started by identifying what data types were held in memory. We then reasoned about the memory layout of those data types to write address arithmetic. In the next section, we will put this reasoning on a more rigorous foundation and extend it to all of System F_U^ω. The types that can be expressed in System F_U^ω are determined by its kind system (Section 6.3). Each proper type is assigned a memory layout (Section 6.4). System F_U^ω's type system assigns types to variables and expressions (Section 6.5). Finally, types and their memory layouts determine how System F_U^ω object accesses are lowered to λ^{L} memory operations (Section 6.6). We have noted that, to manipulate polymorphic unboxed objects, it is necessary to insert code for computing run-time size information and converting objects between storage strategies. This code is explicit in System F_U^ω. It consists of algebraic data type operations and is compiled no differently from other code.

Storage strategy conversions are encoded using algebraic data types, as well. In System F_U^ω, function my_add is given bare integer objects (which have type Stored Int), but the + operator only works on value integer objects (which have type Int), so a conversion from the former to the latter is needed. We give the definition of Stored in Section 6.1, but the essential idea is that a Stored τ is a bare object containing a τ and nothing else, for any value type τ. An Int is retrieved from a Stored Int by reading its contents—this is just a $\mathbf{load}_{\mathsf{Int}}$ operation. A new Stored Int is created by writing an Int into a new object—this is just a $\mathbf{store}_{\mathsf{Int}}$ operation. Similarly to Stored, there is a type constructor Boxed for allocating a bare object on the heap. We refer to type constructors whose purpose is to cast between storage strategies, such as Stored and Boxed, as *adapter types*.

The System F_U^ω code of my_add uses these adapter types to convert between storage strategies. We illustrate the idea here.

$\mathsf{my_add} : \mathsf{Stored\ Int} \to \mathsf{Stored\ Int} \to \mathsf{Boxed\ (Stored\ Int)}$
$\mathsf{my_add}\ p\ q = \mathbf{case}\ p\ \mathbf{of}\ \mathsf{stored}\ m.$
$\quad \mathbf{case}\ q\ \mathbf{of}\ \mathsf{stored}\ n.$
$\quad \mathsf{szStored\ svInt} \Rightarrow \mathsf{boxed\ (Stored\ Int)\ (stored\ Int}\ (m + n))$

In this function, the case expressions load integers from p and q. The data expression stored Int $(m+n)$ produces an initializer that stores the return value into memory. The data expression szStored svInt $\Rightarrow$ boxed (Stored Int) $(\ldots)$ allocates memory for the return value. The latter data expression allocates memory to hold a Stored Int, executing the expression szStored svInt to compute the size of memory to allocate. Of course, this size can be determined statically, and Triolet is indeed able to optimize it to a literal value in λ^{L}. Boxed objects in polymorphic code may have a statically unknown size, in which case the size is computed at run time.

It would be dangerous to allow an arbitrary integer to be passed for an object's size; inconsistent size values would lead to invalid memory loads or stores. System F_U^ω ensures correct use of sizes by encoding the size of a bare object of type τ in an object of type Sz τ. Sz is an algebraic data type containing an Int. The parameter to Sz is a *phantom type* [23]: it is used to categorize Sz objects but does not appear in their contents. The Triolet compiler generates a size-computing function for each algebraic data type based on its memory layout. The types of these functions ensure that it is not possible to accidentally compute or use an incorrect size.

6 Unboxed Polymorphism in System F_U^ω

System F_U^ω (abbreviated henceforth as F_U^ω) is the Triolet compiler's internal language. At the F_U^ω stage, operations that were implicit in Triolet source code (such as class method lookups and storage strategy conversions) have become explicit, but the language is still sufficiently high-level to make transformation easy. Most of the compiler's optimizations are performed on F_U^ω. The syntax of F_U^ω is shown in Figure 3. The primary differences from ordinary F^ω appear in the parts of the language that deal with algebraic data types: data type definitions, case expressions, and data expressions. There are also extra type-level and kind-level constants to describe unboxed data types. This section introduces these features.

Kinds in System F^ω are the "type system of types", used to classify types and exclude invalid type terms from the language. In F_U^ω, kinds also differentiate data types that are handled differently during code generation. The three storage strategies are kinds. The kind out classifies types of pointers to uninitialized or partly initialized objects. The role of these *output pointers* is discussed in Section 6.2. The kind $\mathbb{Z}$ classifies type-level integers, which we use for describing array sizes. Some compiler algorithms are type- and kind-directed: when they encounter a value (or type), they compute its type (or kind) in order to decide what to do with it.

The type language of F_U^ω is that of System F^ω, extended with built-in and user-defined constants. We write type-level integers with a subscript (e.g., 3_{type}) to distinguish them from integer values. Since we do not perform arithmetic on type-level integers, they are merely symbolic constants as far as the compiler is concerned. Int is an unboxed integer type. AsBare and AsBox are primitive type functions [35] that inspect their argument in order to produce a new type. These type functions establish a one-to-one correspondence between boxed types and bare types, reflecting the fact that any object can be represented in a boxed form or bare form. They are used when translating fully boxed Triolet code to unboxed F_U^ω code. Types are erased when lowering F_U^ω to λ^{L} and do not exist at run time.

Unboxed array types are built with the primitive type constructor Arr. An Arr τ π, for any τ and π, consists of τ instances of π laid out consecutively in memory. For instance, an array of 10 integers has type Arr 10_{type} (Stored Int). An array's size need not be a constant. Unboxed arrays are really a building block for more convenient user-level array objects, as we discuss in the next section.

Data type definitions create additional type-level constants. In a data type definition

Type constructors	T	Data constructors	C
Type-level integers	N_{type}	Type variables	$\alpha, \beta, \gamma, \ldots$
Integer literals	N	Value variables	$\ldots, x, y, z$

Kinds	$\kappa, \iota := \mathsf{box} \mid \mathsf{val} \mid \mathsf{bare} \mid \mathsf{out} \mid \mathbb{Z} \mid \kappa \to \kappa$
Types	$\tau, \pi := \alpha \mid \tau\ \tau \mid \forall \alpha : \kappa.\, \tau \mid \lambda \alpha : \kappa.\, \tau \mid \tau \to \tau$
	$\mid T \mid N_{\text{type}}$
	$\mid \mathsf{Int} \mid \mathsf{Arr} \mid \mathsf{Out} \mid \mathsf{AsBare} \mid \mathsf{AsBox}$
Data types	$\mathit{data} := \mathbf{data}\ T\ \overline{\alpha : \kappa} : \kappa\ \mathbf{where}\ C\ \overline{\alpha : \kappa}\ \overline{\tau}$
Expressions	$c, d, e := x \mid N \mid \mathbf{let}\ x = e\ \mathbf{in}\ e$
	$\mid e\ e \mid e\ \tau \mid \lambda x : \tau.\, e \mid \Lambda \alpha : \kappa.\, e$
	$\mid \mathbf{case}\ e\ \mathbf{of}\ \overline{e} \Rightarrow C\ \overline{\alpha}\ \overline{x}.\ e \mid \overline{e} \Rightarrow C\ \overline{\tau}\ \overline{\tau}\ \overline{e}$

Figure 3: Syntax of System F_U^ω.

$\mathbf{data}\ T\ \overline{\alpha : \kappa} : \kappa_T\ \mathbf{where}\ C\ \overline{\beta : \iota}\ \overline{\tau}$, the type constructor definition $T\ \overline{\alpha : \kappa} : \kappa_T$ defines a new *type constructor* T by specifying how to build proper types from it: T may be applied to any choice of type arguments $\overline{\tau_u : \kappa}$ to produce the proper type $T\ \overline{\tau_u}$, which has kind κ_T. The kind given for κ_T, which must be box, bare, or val, determines a data type's default storage strategy. The *data constructor* definition $C\ \overline{\beta : \iota}\ \overline{\tau}$ defines C by specifying what information is contained in an object, given a particular choice of arguments to T. The existential type variables $\overline{\beta : \iota}$ stand for types that are stored in an object, and the field types $\overline{\tau}$ describe the values stored in an object.

The expression language includes F^ω variables, application, and abstraction terms. Integer literals evaluate to value objects. Let expressions bind the result of an expression to a variable while executing another expression. Two forms of expressions access algebraic data types. A *data* expression takes some arguments and assembles them into a new object. A *case* expression reads the fields of an object. F_U^ω extends the conventional, fully boxed formulation of algebraic data types with run-time size information. In the fully boxed formulation, a data expression $C\ \overline{\tau}\ \overline{\pi}\ \overline{e}$ constructs an object of type $T\ \overline{\tau}$ containing the existential types $\overline{\pi}$ and field values computed by $\overline{e}$. In a fully boxed formulation, a case expression $\mathbf{case}\ e\ \mathbf{of}\ C\ \overline{\alpha}\ \overline{x}.\ d$ inspects the object computed by e, binding each type variable α to an existential type and each variable x to a field before evaluating d. Essentially, it extracts the parameters $\overline{\pi}$ and $\overline{e}$ that were passed to an earlier data expression. F_U^ω's data and case expressions additionally take run-time size information that is used to generate address computation. This information is passed as a sequence of extra *size parameters*. For example, in the data expression szStored svInt $\Rightarrow$ boxed (Stored Int) x, the subexpression szStored svInt computes the size parameter. Two or more parameters are written as a comma-separated list, e.g., (x, y). When there are no parameters, we omit the $\Rightarrow$ symbol. The number and type of parameters is determined by an analysis of data type definitions.

For simplicity, we discuss compiler support for algebraic product types only. Triolet supports algebraic sum types, in which data type definitions are generalized to have multiple constructor signatures. We will use sum types from time to time when discussing example uses of data types. We write multiple constructor signatures as alternations: $\mathbf{data}\ T\ \overline{\alpha : \kappa} : \kappa_T\ \mathbf{where}\ \{C_1\ \overline{\beta : \iota_1}\ \overline{\tau_1} \mid C_2\ \overline{\gamma : \iota_2}\ \overline{\tau_2}\}$. Case expressions match a value against one of several alternative patterns, using similar syntax: $\mathbf{case}\ c\ \mathbf{of}\ x \Rightarrow \{C_1\ \overline{\beta}\ \overline{\tau_1}.\ d \mid C_2\ \overline{\gamma}\ \overline{\tau_2}.\ e\}$.

6.1 Some Data Types

Triolet uses algebraic data types extensively. They are used to hold user data, to hold run-time size information, and to effectively change an object's storage strategy. Before entering into a detailed presentation of F_U^ω, we motivate the support for unboxed data type definitions with examples of data types.

Tuples. Probably the simplest interesting example of a parametric data type is a tuple, defined by

$\textbf{data}\ \mathsf{Tuple}\ (\alpha : \mathsf{bare})\ (\beta : \mathsf{bare}) : \mathsf{bare}\ \textbf{where}\ \mathsf{tuple}\ \alpha\ \beta.$

For any given types $\tau : \mathsf{bare}$ and $\pi : \mathsf{bare}$, we can construct the tuple type $\mathsf{Tuple}\ \tau\ \pi : \mathsf{bare}$. Objects of this type contain a value of type τ and a value of type π.

It is also useful to have a tuple type with kind val. We use this type for returning multiple values from a function. Unlike Tuple, a TupleV can be returned in registers or on the stack.

$\textbf{data}\ \mathsf{TupleV}\ (\alpha : \mathsf{val})\ (\beta : \mathsf{val}) : \mathsf{val}\ \textbf{where}\ \mathsf{tupleV}\ \alpha\ \beta$

Adapter Types. Since the fields of a tuple have kind bare, we can only put bare objects into a tuple. For example, Tuple Int Int is not a valid type because the primitive type Int has kind val. Adapter types solve this problem by packaging an object of one storage strategy in an object of another storage strategy. Using the adapter type

$\textbf{data}\ \mathsf{Stored}\ (\alpha : \mathsf{val}) : \mathsf{bare}\ \textbf{where}\ \mathsf{stored}\ \alpha,$

we can construct the type Stored Int representing a bare object containing an Int. A variable holding an Int is likely to become a register holding a machine-level integer. A variable holding an Stored Int, on the other hand, is a pointer to a field of some heap-allocated object. Converting between Int and Stored Int amounts to reading an Int from memory or writing one to memory. This is an extension of an existing technique to package value objects into boxed objects [31].

We use four additional adapter types for converting between bare and box. The primed and un-primed types are structurally identical, but some type functions treat Ref and Boxed differently (Section 6.5.1). These adapter types cover only three of the six possible conversions between box, val, and bare, but they will suffice.

$\textbf{data}\ \mathsf{Ref}\ (\alpha : \mathsf{box}) : \mathsf{bare}\ \textbf{where}\ \mathsf{ref}\ \alpha$

$\textbf{data}\ \mathsf{Boxed}\ (\alpha : \mathsf{bare}) : \mathsf{box}\ \textbf{where}\ \mathsf{boxed}\ \alpha$

$\textbf{data}\ \mathsf{Ref}'\ (\alpha : \mathsf{box}) : \mathsf{bare}\ \textbf{where}\ \mathsf{ref}'\ \alpha$

$\textbf{data}\ \mathsf{Boxed}'\ (\alpha : \mathsf{bare}) : \mathsf{box}\ \textbf{where}\ \mathsf{boxed}'\ \alpha$

Pairs. Fields can be have types more complex than just type variables. For instance, we can pair two values of the same type using a tuple.

$\textbf{data}\ \mathsf{Pair}\ (\alpha : \mathsf{bare}) : \mathsf{bare}\ \textbf{where}\ \mathsf{pair}\ (\mathsf{Tuple}\ \alpha\ \alpha)$

A $\mathsf{Pair}\ \alpha$ has the same memory layout as a $\mathsf{Tuple}\ \alpha\ \alpha$, so we pay no run-time cost for wrapping a tuple in another type.

Option types. A typical example of sum types is the option type, which contains either nothing or a single value. In a fully boxed language, an option object is a boxed object containing a tag value and possibly a pointer to another object. (Some implementations optimize the layout, but they retain at least one pointer.) We can unbox both the option type and its contents:

data Maybe (α : bare) : bare **where** {just α | nothing}

As an example, we can represent a possibly-unbounded interval with the type Pair (Maybe (Stored Int)). The pair holds a lower and upper bound, and a nothing value means that that end of the interval is unbounded. Because all these types are unboxed, an interval is a single block of memory holding two Ints and two tags.

We can define option value types similarly:

data MaybeV (α : val) : val **where** {justV α | nothingV}

An unboxed option value can be held in registers. Provided that space is available, it occupies whatever registers are needed to hold an α plus one register to hold the tag. A function can construct and return a MaybeV object without allocating memory. Some internal Triolet library functions take advantage of this.

Object sizes. The size in words of an object of type τ, for any bare type τ, is represented by an object of type Sz τ (read "size of τ"). Sz objects are produced by size-computing code. Since sizes are just integers, a Sz instance contains just an Int.

data Sz (α : bare) : val **where** sz Int

There are two similar data types for associating integer values with other types. Sv holds the size of a type with kind val. Z holds the integer value corresponding to a type-level integer.

data Sv (α : val) : val **where** sv Int

data Z (ν : $\mathbb{Z}$) : val **where** z Int

The types built with these type constructors are *singleton types*. Objects of a given type τ can have only one possible size, and so objects of type Sz τ can have only one possible value.

Run-time representations. A few pieces of data are very useful for manipulating polymorphic bare objects. For instance, it is often useful to copy a bare object from one place to another. There is no way to copy a bare object of unknown type, so this functionality can only be provided by passing a suitable "copy" function to polymorphic code. For a given type α, we package frequently used data into objects of type Rep α.

data Rep (α : bare) : box
where rep (Sz α)
    (α → Out α → Store)
    (AsBox α → Out α → Store)
    ((Out α → Store) → AsBox α)

The first field is the size of an α. The second field is a "copy" function. Given a parameter of type α, a copy function returns an initializer of type Out α → Store that writes a copy of the parameter to a new location. The third and fourth fields convert between initializers and AsBox α objects. AsBox α is the boxed representation of α, and is isomorphic to α. We define the functions size, copy, asBox, and asBare to extract individual fields of a Rep object.

Delayed functions. Delayed functions are an instructive example of how existential types can be used for encapsulation. A delayed function call consists of a function of type $\beta \to \alpha$ paired with an argument of type β. The call can be executed by passing the argument to the function.

data $\mathsf{Delayed}\ (\alpha : \mathsf{bare}) : \mathsf{bare}$ **where** $\mathsf{delayed}\ (\beta : \mathsf{box})\ \beta\ (\beta \to \alpha)$

The function's return type α is a parameter of $\mathsf{Delayed}$, while its parameter type β is an existential type. Accordingly, a value of type $\mathsf{Delayed}\ \tau$ contains a function with a *known* return type τ but a *hidden* parameter type. The parameter type is known when a $\mathsf{Delayed}$ object is created, so it is certain that the first and second fields have compatible types. However, code that uses a $\mathsf{Delayed}$ object cannot examine the actual type bound to β.

While none of our applications use $\mathsf{Delayed}$, more complex uses of existential types arise in Triolet library code. Iterators are existentially typed so that the user of an iterator need not be aware of the iterator's internal state.

Arbitrary-length arrays. It is common for programming languages to have arbitrary-length array types, as opposed to known-length array types constructed with Arr. We can define such an array as an algebraic data type. The following is a one-dimensional array, $\mathsf{Array1}$.

data $\mathsf{Array1}\ (\alpha : \mathsf{bare}) : \mathsf{bare}$
where $\mathsf{array1}\ (\nu : \mathbb{Z})\ (\mathsf{Z}\ \nu)\ (\mathsf{Boxed}\ (\mathsf{Arr}\ \nu\ \alpha))$

The existential type variable ν holds the hidden array length. To allow programs to inspect the length as an integer value, it is also stored as a field of type $\mathsf{Z}\ \nu$. The real array is stored in a boxed object.

We define a two-dimensional array similarly, using nested Arrs to store the array contents.

data $\mathsf{Array2}\ (\alpha : \mathsf{bare}) : \mathsf{bare}$
where $\mathsf{array2}\ (\mu : \mathbb{Z})\ (\nu : \mathbb{Z})\ (\mathsf{Z}\ \mu)\ (\mathsf{Z}\ \nu)\ (\mathsf{Boxed}\ (\mathsf{Arr}\ \mu\ (\mathsf{Arr}\ \nu\ \alpha)))$

$\mathsf{Arr}\ \mu\ (\mathsf{Arr}\ \nu\ \alpha)$ is the type of an array of arrays of α. It is laid out as a single block of memory, like an array of arrays in C.

To emulate the boxing of array elements that occurs in some programming languages, we can define a boxed array type. A boxed array is an array of pointers to boxed objects.

data $\mathsf{BArray1}\ (\alpha : \mathsf{box}) : \mathsf{bare}$
where $\mathsf{barray1}\ (\nu : \mathbb{Z})\ (\mathsf{Z}\ \nu)\ (\mathsf{Boxed}\ (\mathsf{Arr}\ \nu\ (\mathsf{Ref}'\ \alpha)))$

The type $\mathsf{Ref}'\ \alpha$ is a pointer to an α. In performance comparisons, we contrast the performance of boxed and unboxed array types by writing programs that use either $\mathsf{BArray1}$ or $\mathsf{Array1}$. The Triolet library contains overloaded operations that work on either array type.

Higher-kinded types. Data types can be parameterized over type operators. Such types are called *higher-kinded* by analogy with higher-order functions. In Triolet, class dictionaries of higher-kinded type classes are higher-kinded data types.

As a simple example of a higher-kinded type, here is a tree type.

data $\mathsf{Tree}\ (\gamma : \mathsf{bare} \to \mathsf{bare})\ (\alpha : \mathsf{bare}) : \mathsf{box}$ **where** $\mathsf{tree}\ \alpha\ (\gamma\ (\mathsf{Ref}\ (\mathsf{Tree}\ \gamma\ \alpha)))$

Parameter γ determines how the children of each tree node are organized. Parameter α determines how nodes are labeled. Instantiating γ with Maybe gives each node zero or one children, making a singly-linked list. Instantiating γ with $\mathsf{Array1}$ gives each node a variable-length array of children, making a rose tree.

6.2 Execution Semantics

F_U^ω is a functional language. All data are immutable, and functions and expressions typically have no effect other than to produce a return value. There are two complications that we must address, namely side effects and imperative assignment.

Side effects arise in F_U^ω because functions are partial. A function typically returns a value, but could also enter an infinite loop or signal a run-time error. F_U^ω has an imprecise side effect semantics: the compiler does not preserve the order of side-effecting operations and may remove side effects. A side effect free function is never transformed into a side effecting one. The freedom to transform side effects somewhat resembles a C compiler's ability to reorder code that has undefined behavior.

An imperative operation, such as writing to memory, observes the state of memory produced by earlier operations and transforms memory to a new state that is visible to later operations. Because a memory operation may influence a later operation, the execution order of memory operations should be preserved in a compiler. We use a store-passing encoding to explicitly express the ordering of imperative operations. The types given to data expressions and imperative operations are best understood in terms of store-passing, which we summarize here.

In a store-passing encoding, a *store* is a value representing a particular state of memory [32, 8]. Our encoding is fine-grained: a store represents a part of memory, and multiple stores representing disjoint parts of memory can exist simultaneously. Semantically, a store is a mapping from addresses to values. A store maps address a to value v iff it describes a state of memory that includes the value v stored at address a.

Imperative functions transform an old state of memory to a new state. Encoded as store-passing, such a function takes store parameters representing the old state and returns a new store representing the new state. When a program performs a sequence of accesses to the same store, each access returns a store that is passed as a parameter to the next, ensuring that the compiler preserves execution order. Since F_U^ω is generated from a functional program that never overwrites objects, each object is written exactly once in the majority of cases. This unique write does not take an input store, since there is no previous memory access whose execution order must be preserved. Some internal library functions imperatively update data, and they take input stores. Stores are purely a compile-time notion to enforce ordering. At run time, stores are values of type Store that contain no data. Fine-grained stores are compatible with parallel execution: threads can imperatively operate on their own private stores without risk of race conditions.

At a given time, a given address can be part of an immutable object or a mutable store, but not both. Modifying a store does not affect the value of any immutable object. Data expressions allocate memory, run initializers to write the allocated memory, and reinterpret the initialized stores as immutable objects. A pointer to an imperatively modified object of type τ has type $\mathsf{Out}\ \tau$. An initializer has type $\mathsf{Out}\ \tau \to \mathsf{Store}$, signifying that it is a function that takes a pointer to an uninitialized object of type τ and creates a store containing an initialized object at that address. Initializers may take additional parameters; by convention the output pointer is the last one.

If we wished to utilize our store-passing model to verify memory safety or analyze the memory access behavior of F_U^ω programs, we would not lump all stores into the type Store, but rather distinguish them by their contents. We would also take care to enforce linearity. However, our only use for stores is to record ordering constraints generated by the compiler or manually inserted into the internal library. A simple store type is adequate for this purpose.

$$\frac{\alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau : \kappa_1 \to \kappa_2 \qquad \Gamma \vdash \pi : \kappa_1}{\Gamma \vdash \tau\, \pi : \kappa_2}$$

$$\frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2 \qquad \kappa_2 \in \{\mathsf{box}, \mathsf{val}, \mathsf{bare}\}}{\Gamma \vdash \forall \alpha : \kappa_1 .\, \tau : \kappa_2} \qquad \frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \lambda \alpha : \kappa_1 .\, \tau : \kappa_1 \to \kappa_2}$$

$$\frac{\begin{array}{ll}\Gamma \vdash \tau : \kappa_1 & \kappa_1 \in \{\mathsf{box}, \mathsf{val}, \mathsf{bare}, \mathsf{out}\} \\ \Gamma \vdash \pi : \kappa_2 & \kappa_2 \in \{\mathsf{box}, \mathsf{val}, \mathsf{bare}, \mathsf{out}\}\end{array}}{\Gamma \vdash \tau \to \pi : \mathsf{box}}$$

$$\begin{array}{ll}\mathsf{Int} : \mathsf{val} & \mathsf{Arr} : \mathbb{Z} \to \mathsf{bare} \to \mathsf{bare} \\ \mathsf{Out} : \mathsf{bare} \to \mathsf{out} & \mathsf{AsBare} : \mathsf{box} \to \mathsf{bare} \\ \mathsf{AsBox} : \mathsf{bare} \to \mathsf{box} & \end{array}$$

Figure 4: Kinding rules and kinds of constants in F_U^ω.

6.3 Kind System

F_U^ω generalizes the kind system of F^ω to support multiple storage strategies as shown in Figure 4. The difference from F^ω is that, where F^ω has only the kind $\star$ of proper types, F_U^ω has the four kinds box, val, bare, and out. The kinding rules statically assign one of these kinds to every variable, expression, and object field. Kind information is used during optimization and code generation.

Functions are boxed and can take and return objects of any kind. Since universal quantification has no run-time effect, a universally quantified type $\forall \alpha : \kappa . \tau$ has the same kind as τ. Objects of kind box, val, and bare can have universally quantified types. An output pointer is always associated with a bare object of a known type and its type cannot vary independently of the bare object's type. Consequently, it does not seem useful to allow objects of kind out to have a universally quantified type, and we do not allow it. Other kinding rules are the same as in F^ω.

Kinding rules are also applied to data type definitions. Consider a data type definition, **data** $T\ \overline{\alpha : \kappa} : \kappa_T$ **where** $C\ \overline{\beta : \iota}\ \overline{\tau}$. The data type's storage strategy κ_T must be box, bare, or val. Furthermore, the field types $\overline{\tau}$ must each have kind box, bare, or val. The data type definition introduces a type constructor T with kind $\overline{\kappa} \to \kappa_T$.

Additional constraints are imposed on some data type definitions. Because objects with the val storage strategy are meant to fit in registers, we only allow their fields to have kind val (since value types fit in registers) or box (pointers, which fit in registers). Unboxed data types may not have recursive definitions. If unboxed recursive data type definitions were permitted, we could define a dubious data type that contains an unboxed instance of itself, such as the following.

data ArrayList : bare **where** arrayList Int ArrayList

To avoid such situations, recursively defined types must be boxed.

6.3.1 Type Environments

Type environments ascribe kinds to type variables and type constructors, types to variables, and type signatures to data constructors.

$$\Gamma := \overline{t}$$
$$t := \alpha : \kappa \mid T : \kappa \mid x : \tau \mid C : \forall \overline{\alpha : \kappa} .\, (\overline{\tau_s})(\overline{\pi_s}) \Rightarrow \forall \overline{\beta : \iota} .\, \overline{\tau} \to T\ \overline{\alpha}$$

The first three type ascriptions are conventional, but the last merits explanation.

In a fully boxed language, a data constructor has a type signature of the form $\forall \overline{\alpha : \kappa}.$ $\forall \overline{\beta : \iota}. \overline{\tau} \to T\ \overline{\alpha}$. It looks like a function type, which reflects the similarity between data constructors and functions: one can think of a data constructor as a built-in function for creating objects. The type parameters $\overline{\alpha : \kappa}, \overline{\beta : \iota}$, value parameters $\overline{\tau}$, and type constructor T are taken directly from the data type definition where C is defined. It helps to think of the type as having two parts. The part $\forall \overline{\alpha : \kappa}.\ \ldots \to T\ \overline{\alpha}$ comes from the type constructor's definition and describes the types of objects constructed by data expressions using C. The other part, $\forall \overline{\beta : \iota}.\ \overline{\tau}$, comes from the data constructor's definition and describes the contents of those objects.

F_U^ω adds to the fully boxed type signature two lists of types needed for implementing unboxed polymoprhism. We write the extra information as $(\overline{\tau_s})(\overline{\pi_s}) \Rightarrow$ in a constructor signature. When both lists are empty, we omit them from the signature. The size parameter types $\overline{\tau_s}$ are the types of objects that are passed to case or data expressions for run-time size computation. Size parameter types are Sz, Sv, or Z types. The *static types* $\overline{\pi_s}$ are types that must have a statically known layout whenever the constructor signature is used. These lists of types are generated by analyzing data type definitions (Section 6.4.3). Note that, while these types play a role in code generation, they do not change the high-level meaning of unboxed data types. Unboxed data types contain the same information as fully boxed types.

A constructor signature can be instantiated to a particular choice of universal type parameters. We write $\Gamma \vdash C \geq sig$ to mean that C's signature is taken from Γ and instantiated to the signature sig. For a constructor $C : \forall \overline{\alpha : \kappa}. (\overline{\tau_s})(\overline{\pi_s}) \Rightarrow \forall \overline{\beta : \iota}. \overline{\tau} \to T\ \overline{\alpha}$ and any well-kinded substitution $\theta = [\overline{\pi / \alpha}]$, we have $C \geq (\overline{\theta(\tau_s)})(\overline{\theta(\pi_s)}) \Rightarrow \forall \overline{\beta : \iota}.$ $\overline{\theta(\tau)} \to T\ \overline{\pi}$. Instantiation is used to produce a data constructor signature matching a given type: a data constructor signature for a given type $T\ \overline{\pi}$ is obtained by instantiating T's data constructor's signature with $\overline{\pi}$.

6.4 Memory Layout

An object is represented concretely as a sequence of primitive values. After compilation, these values may be held in registers or laid out sequentially in memory. To generate code that creates, reads, or writes an object, we must know its structure in terms of primitive values, which we call its *layout*. This section defines layouts and associates F_U^ω types with layouts. The relationship between types and layouts will be used to construct algorithms for computing run-time type information.

Layouts are defined by

$$\varsigma := \mathsf{Int} \mid \mathsf{Ptr} \mid \langle\rangle \mid \varsigma \times \varsigma.$$

The primitive layouts Int and Ptr occupy one word of memory. The unit layout $\langle\rangle$ occupies zero words of memory. Data structures consisting of more than one primitive value have product layouts. The product layout $\varsigma_1 \times \varsigma_2$ consists of a ς_1 abutting a ς_2, like a two-member `struct` in C. We abbreviate nested products $\varsigma_1 \times (\varsigma_2 \times \cdots)$ as $\langle \varsigma_1, \varsigma_2, \ldots \rangle$ or $\langle \overline{\varsigma} \rangle$. An unboxed array is a product of N instances of ς, written ς^N. We write $\mathrm{size}(\varsigma)$ for the size in words of ς. The layouts that we use here are simplified from Triolet's layouts; they do not allow primitive types of different sizes and alignments, sum types, or tags on objects. More general layouts entail more elaborate computation in some places, but do not affect the overall framework.

An object is laid out as a sequence of fields, which we write as a product. Since boxed fields are just pointers, an object with two boxed fields would have layout $\langle \mathsf{Ptr}, \mathsf{Ptr} \rangle$. An object's unboxed fields become part of the object's layout. For example, the layout of a $\mathsf{Stored\ Int}$ is $\langle \mathsf{Int} \rangle$, rather than $\langle \mathsf{Ptr} \rangle$, since its field is unboxed. A $\mathsf{Tuple\ (Stored\ Int)\ (Stored\ Int)}$ has two unboxed fields, and each field has layout

$$\frac{\Gamma \vdash \tau : \mathsf{box}}{\Gamma \vdash \tau \downarrow_{\text{field}} \mathsf{Ptr}} \qquad \frac{\Gamma \vdash \tau : \kappa \qquad \kappa \in \{\mathsf{val}, \mathsf{bare}\} \qquad \Gamma \vdash \tau \downarrow \varsigma}{\Gamma \vdash \tau \downarrow_{\text{field}} \varsigma}$$

$$\frac{N \propto N_{\text{type}} \qquad \Gamma \vdash \tau \downarrow_{\text{field}} \varsigma}{\Gamma \vdash \mathsf{Arr}\ N_{\text{type}}\ \tau \downarrow \varsigma^N} \qquad \frac{}{\Gamma \vdash \mathsf{Int} \downarrow \mathsf{Int}}$$

$$\frac{\Gamma \vdash C \geq (\overline{\tau_s})(\overline{\pi_s}) \Rightarrow \forall \overline{\beta : \iota}.\, \overline{\tau} \to T\ \overline{\pi} \qquad \Gamma, \overline{\beta : \iota} \vdash \overline{\tau \downarrow_{\text{field}} \varsigma}}{\Gamma \vdash T\ \overline{\pi} \downarrow \langle \overline{\varsigma} \rangle}$$

$$\frac{\Gamma, \alpha : \kappa \vdash \tau \downarrow \varsigma}{\Gamma \vdash \forall \alpha : \kappa.\, \tau \downarrow \varsigma}$$

Figure 5: Layouts of F_U^ω types.

$\langle \mathsf{Int} \rangle$, so the tuple's layout is $\langle \langle \mathsf{Int} \rangle, \langle \mathsf{Int} \rangle \rangle$. Note that this layout has the same memory representation as $\langle \mathsf{Int}, \mathsf{Int} \rangle$.

Some polymorphic types cannot meaningfully be assigned a layout. Objects with such *ambiguous* layouts cannot be manipulated in F_U^ω. Consider, for instance, the problem of allocating a bare object with an ambiguous layout. The layout tells us how much memory the object occupies; if we can't assign it a layout, we can't decide how much memory to allocate. Ambiguity can occur with existential types. Suppose we had an object of type Exists, defined by **data** Exists : bare **where** exists α : bare α. The existentially bound type α determines the layout of the object's contents; different Exists objects could have different layouts despite having the same type. Universally quantified types can also have an ambiguous layout. Suppose we had an object of type $\forall \alpha : \mathsf{bare}.\, \alpha$. This is an object that can be instantiated to any bare type. By instantiating the object to different types, we can view it as having any layout we choose! This is not safe because only one layout of data can actually exist in memory.

On the other hand, there are many situations where polymorphic objects do not have an ambiguous layout. Although the arbitrary-length array type Array1 from Section 6.1 contains an existentially typed array, Array1 has an unambiguous layout. The unknown-length array field is boxed, so even though the field has an unknown type, its layout is Ptr. A type has an ambiguous layout only when it contains an unboxed field with an unknown type. This situation is easily detectable when computing layouts, as we describe below.

It is not actually possible to introduce a universal quantifier in a way that produces an object with an ambiguous layout. There is no such thing as an object with a parametrically polymorphic layout; any time an object is created, enough run-time type information is passed in to uniquely determine its layout. However, it is possible to define a data type with an ambiguous layout. Such definitions are treated as compile-time errors.

6.4.1 Memory Layout of F_U^ω Types

We formalize memory layout in the layout rules of Figure 5. We will use this definition of layouts when formulating code generation algorithms, including generation of run-time size computation and memory operations for accessing object fields. We describe the layout of a type using a judgment of the form $\Gamma \vdash \tau \downarrow \varsigma$, meaning that in type environment Γ, an object of type τ has layout ς. The layout rules can be read as an algorithm for computing ς from Γ and τ. As an algorithm, layout computation proceeds by structural recursion on τ and on the fields of objects. The related judgment

$\Gamma \vdash \tau \downarrow_{\text{field}} \varsigma$ means that an object field of type τ has layout ς. The first two rules state that the layout of a boxed field is Ptr, while the layout of an unboxed field of type τ is the layout of τ.

The remaining rules give the layouts of types. The layout of an array $\mathsf{Arr}\ N_{\text{type}}\ \tau$ is computed by finding the layout of τ, then creating a product of N instances of that layout. We write $N \propto N_{\text{type}}$ to mean that the type-level integer N_{type} represents the same number as the integer N. The layout of Int is simply Int. The next rule gives the layout of any algebraic data type $T\ \overline{\pi}$. The layout of the type is the product $\langle \overline{\varsigma} \rangle$ of the layouts of its fields. The field types, found from the type parameters and constructor type signature, determine the field layouts. The final rule gives the layout of a universally quantified type. Since universal quantification has no run-time effect, the layout of $\forall \alpha : \kappa.\ \tau$ is the same as the layout of τ.

Types that have an ambiguous layout are not assigned a layout by the rules of Figure 5. When a type has an ambiguous layout, attempting to derive the layout produces an unsolvable premise such as $\Gamma \vdash \alpha \downarrow \varsigma$, where we need the layout of a type variable in order to finish computing the layout. There is no rule to get the layout of a type variable; after all, an unknown type has an unknown layout. To decide whether a type has an ambiguous layout, we can simply attempt to derive its layout. For instance, attempting to derive the layout of $\forall \alpha : \mathsf{bare}.\ \alpha$ would proceed as follows.

$$\frac{\alpha : \mathsf{bare} \vdash \alpha \downarrow \varsigma}{\vdash \forall \alpha : \mathsf{bare}.\ \alpha \downarrow \varsigma}$$

Derivation cannot proceed past this point, and this indicates that the type has an ambiguous layout. In contrast, a layout can be derived for $\forall \alpha : \mathsf{bare}.\ \mathsf{Sz}\ \alpha$ since objects of this type do not actually contain an object of type α.

$$\frac{\dfrac{\Gamma, \alpha : \mathsf{bare} \vdash \mathsf{sz} : \mathsf{Int} \rightarrow \mathsf{Sz}\ \alpha \qquad \dfrac{\overline{\Gamma, \alpha : \mathsf{bare} \vdash \mathsf{Int} \downarrow \mathsf{Int}}}{\Gamma, \alpha : \mathsf{bare} \vdash \mathsf{Int} \downarrow_{\text{field}} \mathsf{Int}}}{\Gamma, \alpha : \mathsf{bare} \vdash \mathsf{Sz}\ \alpha \downarrow \langle \mathsf{Int} \rangle}}{\Gamma \vdash \forall \alpha : \mathsf{bare}.\ \mathsf{Sz}\ \alpha \downarrow \langle \mathsf{Int} \rangle}$$

The Triolet compiler detects existential data type definitions with ambiguous layouts in this way and reports them as errors. Ambiguous universal types are detected when type checking functions.

6.4.2 Dynamic Layout Computation

We cannot use Figure 5 to directly compute layouts in polymorphic code, because types are not determined until a function executes. For instance, a compiler may only know that a variable has type $\mathsf{Tuple}\ \alpha\ \beta$, where α and β are type parameters of the enclosing function. Although the layouts of α and β cannot be derived at compile time, they can be derived at run time by creating, for each statically unknown type, a data structure that holds layout information for that type. This is similar to type passing [18]. Triolet does *not* do this, but it computes run-time type information in a similar way.

To more clearly establish the link between layouts and run-time type information, we present a way of dynamically computing layouts. We encode layouts as an algebraic data type, L, and use three additional type definitions to associate a layout with a type.

$\textbf{data}\ \mathsf{L} : \mathsf{box}\ \textbf{where}\ \{\mathsf{intL} \mid \mathsf{ptrL} \mid \mathsf{unitL} \mid \mathsf{prodL\ L\ L}\}$
$\textbf{data}\ \mathsf{Layout}\ (\alpha : \mathsf{bare}) : \mathsf{box}\ \textbf{where}\ \mathsf{layout\ L}$
$\textbf{data}\ \mathsf{LayoutV}\ (\alpha : \mathsf{val}) : \mathsf{box}\ \textbf{where}\ \mathsf{layoutV\ L}$
$\textbf{data}\ \mathsf{LayoutP}\ (\alpha : \mathsf{box}) : \mathsf{box}\ \textbf{where}\ \mathsf{layoutP\ L}$

Given an L, we can compute its size by summing the sizes of its components.

$\mathsf{sizeOfL} : \mathsf{L} \to \mathsf{Int}$
$\mathsf{sizeOfL}\ x = \textbf{case}\ x\ \textbf{of}\ \{\mathsf{intL}.\ 1 \mid \mathsf{ptrL}.\ 1 \mid \mathsf{unitL}.\ 0 \mid \mathsf{prodL}\ y\ z.\ \mathsf{sizeOfL}\ y + \mathsf{sizeOfL}\ z\}$

The size of a Layout, LayoutV, or LayoutP is the size of its L field.

We translate Figure 5 to executable code by rephrasing layout judgments as typing judgments on layout-computing expressions. A layout judgment $\Gamma \vdash \tau \downarrow \varsigma$ becomes a typing judgment $\Gamma \vdash e : \mathsf{Layout}\ \tau$ (or $\mathsf{LayoutV}\ \tau$ or $\mathsf{LayoutP}\ \tau$, depending on τ's kind). The proposition $N \propto N_{\text{type}}$ becomes the typing judgment $\Gamma \vdash e : \mathsf{Z}\ N_{\text{type}}$.

The translation turns axioms into constants and inference rules into functions. For instance, the axiom $\dfrac{}{\Gamma \vdash \mathsf{Int} \downarrow \mathsf{Int}}$ indicates that type Int has layout intL. We first rephrase it as a typing judgment.

$$\frac{}{\Gamma \vdash \mathsf{layoutV\ Int\ intL} : \mathsf{LayoutV\ Int}}$$

Then, we phrase it as code by defining a global constant.

$\mathsf{layoutInt} : \mathsf{LayoutV\ Int}$
$\mathsf{layoutInt} = \mathsf{layoutV\ Int\ intL}$

The inference rule for the algebraic data type Stored is as follows.

$$\frac{\Gamma \vdash \mathsf{stored} : ()(\tau) \Rightarrow \tau \to \mathsf{Stored}\ \tau \qquad \dfrac{\Gamma \vdash \tau \downarrow \varsigma}{\Gamma \vdash \tau \downarrow_{\text{field}} \varsigma}}{\Gamma \vdash \mathsf{Stored}\ \tau \downarrow \langle \varsigma \rangle}$$

We discard the stored constructor signature, which carries no useful information, and generate a rule to compute the layout of Stored τ from the layout of τ.

$$\frac{\Gamma \vdash \mathsf{layoutV}\ \tau\ e : \mathsf{LayoutV}\ \tau}{\Gamma \vdash \mathsf{layout}\ (\mathsf{Stored}\ \tau)\ (\mathsf{prodL}\ e\ \mathsf{unitL}) : \mathsf{Layout}\ (\mathsf{Stored}\ \tau)}$$

We can instantiate this rule to any well-typed choice of τ and e. The most general choice is to let the unknowns be variables.

$$\frac{\Gamma, \alpha : \mathsf{val}, x : \mathsf{L} \vdash \mathsf{layoutV}\ \alpha\ x : \mathsf{LayoutV}\ \alpha}{\Gamma, \alpha : \mathsf{val}, x : \mathsf{L} \vdash \mathsf{layout}\ (\mathsf{Stored}\ \alpha)\ (\mathsf{prodL}\ x\ \mathsf{unitL}) : \mathsf{Layout}\ (\mathsf{Stored}\ \alpha)}$$

We turn the generalized rule into a global function that extracts x from its argument and uses it to build a new Layout object.

$\mathsf{layoutStored} : \forall \alpha : \mathsf{val}.\ \mathsf{LayoutV}\ \alpha \to \mathsf{Layout}\ (\mathsf{Stored}\ \alpha)$
$\mathsf{layoutStored}\ \alpha\ y = \textbf{case}\ y\ \textbf{of}\ \mathsf{layoutV}\ x.\ \mathsf{layout}\ (\mathsf{Stored}\ \alpha)\ (\mathsf{prodL}\ x\ \mathsf{unitL})$

Such functions can be used to dynamically compute layouts. Supposing that Triolet used L objects for run-time size information, we could use layoutStored to compute layout information for accessing a tuple based on its run-time type as shown below.

$\mathsf{secondVal} : \forall\alpha:\mathsf{val}.\ \mathsf{LayoutV}\ \alpha \to \mathsf{Tuple}\ (\mathsf{Stored}\ \alpha)\ (\mathsf{Stored}\ \alpha)$
$\mathsf{secondVal}\ \alpha\ x\ y = \mathbf{let}\ s = \mathsf{layoutStored}\ \alpha\ x\ \mathbf{in}$
$\quad\quad\quad\quad \mathbf{case}\ y\ \mathbf{of}\ (s, s) \Rightarrow \mathsf{tuple}\ u\ v.\ v$

This function takes the layout of α as a run-time parameter and calls layoutStored to compute the layout of Stored α. This computed layout is passed to the case expression, which uses it to compute the size of each tuple field.

6.4.3 Computing Data Constructor Type Signatures

Data constructor type signatures contain two lists of types, the size parameter types and static types, that are generated by an analysis of data type definitions. These lists indicate, respectively, what run-time and compile-time information is needed in order to compute the layout of a data type. The analysis follows the same steps as if it were computing a data type's layout. Whenever an unknown type is reached, that type is recorded.

Suppose we have the data type definition **data** S (α : val) (β : bare) : box **where** s α (Tuple β (Stored Int)). The analysis begins by computing the types of all fields of an object of type S α β. These types can be read directly from the data constructor definition. Boxed fields are ignored. The unboxed field types are partitioned into those with kind bare and those with kind val. There is one bare field, with type Tuple β (Stored Int). The analysis attempts to compute the layout of this field, and collects a list of all types whose layouts cannot be computed. In this case, β is the only such type. The size of β must be passed at run time so that F^{ω}_{U} code can compute the layout of an S object. The size's type, Sz β, is put into the list of static types. In this example, this size is the only thing in the list. There is one val field, with type α. Again, the analysis attempts to compute the layout of this field and collects a list of all types whose layouts cannot be computed. The resulting list, which contains α, becomes the list of static types. The constructor signature, $\mathsf{s} : \forall\alpha{:}\mathsf{val}.\forall\beta{:}\mathsf{bare}.(\mathsf{Sz}\ \beta)(\alpha) \Rightarrow \alpha \times \mathsf{Tuple}\ \beta\ (\mathsf{Stored\ Int}) \to \mathsf{S}\ \alpha\ \beta$, is assembled from these two parameter lists and parts of the data type definition.

Several special cases are worth mentioning. If a data type has no unboxed fields, then $\overline{\tau_s}$ and $\overline{\pi_s}$ are both empty; no run-time size information is required and there are no constraints on polymorphism. Thus, fully boxed types are a special case of F^{ω}_{U} types. Also, as we mentioned in Section 6.3, value types cannot have bare fields. Consequently, an algebraic value type always has empty $\overline{\tau_s}$, and run-time size information is not required to access it.

6.5 Type System

The type system of F^{ω}_{U} is adapted from F^{ω} with algebraic data types. Type functions AsBox and AsBare are introduced. Restrictions are added so that value variables have a statically known layout, and so that objects with ambiguous layouts cannot be created or used. Data expressions are given an unconventional typing rule to reflect the use of initializers.

6.5.1 Relating Storage Strategies with Type Functions

There are various ways of converting between bare and boxed types. For instance, suppose we wanted to convert an object of type Ref (Int → Int) to a boxed object. We could consider creating a Boxed (Ref (Int → Int)), which is a boxed object containing a reference. However, Int → Int is also a boxed type, and a better choice since it involves fewer levels of pointer indirection. In this case, it is clear that the Ref constructor can be removed, but the decision is not so obvious when compiling polymorphic code. A

variable in a polymorphic function may have type $\mathsf{Ref}\ (\mathsf{Int} \to \mathsf{Int})$ in one execution of the function, but type $\mathsf{Stored\ Int}$ in another; now we cannot statically decide whether there is a Ref constructor to remove.

Triolet dynamically decides how to convert between bare and boxed types. The built-in type functions $\mathsf{AsBare} : \mathsf{box} \to \mathsf{bare}$ and $\mathsf{AsBox} : \mathsf{bare} \to \mathsf{box}$ describe, within the type system, how types are converted. A bare type is converted to a boxed type in one of two ways: if it has the form $\mathsf{Ref}\ \pi$, then the Ref adapter is removed; otherwise, the Boxed adapter is applied. Similarly, to convert from a boxed to a bare type, the Boxed adapter is removed if possible, otherwise Ref is applied. This is expressed by the definitions below.

$$\mathsf{AsBox}\ \tau = \begin{cases} \pi & \text{if } \tau = \mathsf{Ref}\ \pi \\ \mathsf{Boxed}\ \tau & \text{otherwise} \end{cases}$$

$$\mathsf{AsBare}\ \tau = \begin{cases} \pi & \text{if } \tau = \mathsf{Boxed}\ \pi \\ \mathsf{Ref}\ \tau & \text{otherwise} \end{cases}$$

We take these type functions as a specification of what should happen when converting between bare and boxed types. Section 7.2 defines conversions that conform to these types.

These functions should be inverses; that is, if we convert a type from bare to boxed and back to bare, we should get the type we started with. For this to be true, we require that a type of the form $\mathsf{Ref}\ (\mathsf{Boxed}\ \tau)$ or $\mathsf{Boxed}\ (\mathsf{Ref}\ \tau)$ is never created. These types would violate the inverse property: $\mathsf{AsBare}\ (\mathsf{AsBox}\ (\mathsf{Ref}\ (\mathsf{Boxed}\ \tau))) = \tau \neq \mathsf{Ref}\ (\mathsf{Boxed}\ \tau)$. To prevent the creation of such types, Ref and Boxed are only introduced by reduction on AsBare or AsBox. That is, a type of the form $\mathsf{Ref}\ \tau$ is permitted only where $\mathsf{AsBare}\ \tau$ would reduce to it, and similarly for Boxed.

6.5.2 Restrictions on Layouts

Variables of kind val must have a statically known layout. F_U^ω's typing rules include constraints to ensure that val objects can only be created or used if their layout is statically known. Note that this is not a restriction on types, but on *uses* of types. For instance, the lambda expression $\Lambda\alpha : \mathsf{val}.\ \lambda x : \alpha.\ 0$ is ill-typed because the value variable x has an unknown layout; but the type of that lambda expression, $\forall\alpha : \mathsf{val}.\ \alpha \to \mathsf{Int}$, is valid. We write τ ok to mean that, if the proper type τ has kind val, then it has a statically known layout.

$$\frac{\Gamma \vdash \tau : \mathsf{val} \qquad \Gamma \vdash \tau \downarrow \varsigma}{\Gamma \vdash \tau \text{ ok}} \qquad\qquad \frac{\Gamma \vdash \tau : \kappa \qquad \kappa \in \{\mathsf{bare}, \mathsf{box}, \mathsf{out}\}}{\Gamma \vdash \tau \text{ ok}}$$

Variables must have types that are ok, and expressions must return types that are ok. To add these conditions to every typing rule would be redundant; it is sufficient to check the variables introduced by abstraction and case expressions, and the objects returned by application and data expressions. The abstraction and application rules are shown below. (Case and data expressions are discussed in Section 6.5.4.)

$$\frac{\Gamma \vdash \tau \text{ ok} \qquad \Gamma, x : \tau \vdash e : \pi}{\Gamma \vdash \lambda x : \tau.\ e : \tau \to \pi} \qquad\qquad \frac{\Gamma \vdash c : \tau \to \pi \qquad \Gamma \vdash e : \tau \qquad \Gamma \vdash \pi \text{ ok}}{\Gamma \vdash c\ e : \pi}$$

$$\frac{\Gamma \vdash e : \forall\alpha : \kappa.\ \tau \qquad \Gamma \vdash \pi : \kappa \qquad \Gamma \vdash \tau[\pi/\alpha] \text{ ok}}{\Gamma \vdash e\ \pi : \tau[\pi/\alpha]}$$

6.5.3 Typing Fully Boxed Algebraic Data Types

Before going into how case and data expressions are typed in F_U^ω, we review how they are typed in a conventional, fully-boxed language. A case expression **case** e **of** $p.\ c$ reads the fields of the object returned by e, binding them to the variables in the pattern p, so that they can be used by c.

$$\frac{\Gamma \vdash C \geq \forall \overline{\beta : \iota}.\, \overline{\tau} \to T\ \overline{\pi} \qquad \Gamma \vdash e : T\ \overline{\pi} \qquad \Gamma, \overline{\beta : \iota}, \overline{x : \tau} \vdash c : \pi_c}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ C\ \overline{\beta}\ \overline{x}.\ c : \pi_c}$$

The data constructor C is instantiated to a signature matching the type of the scrutinee expression e. This signature tells us what are the contents of an object of this type. The kinds $\overline{\iota}$ and types $\overline{\tau}$ of the pattern-bound variables are added to the type environment of the case expression body c.

A data expression creates a new object. The given universal and existential type parameters, $\overline{\tau_u}$ and $\overline{\tau_e}$ must match the kinds given in the constructor's type signature. The type parameters determine the types of the given field expressions $\overline{e}$. The data constructor and universal type paramters determine the data expression's type.

$$\frac{\Gamma \vdash C \geq \forall \overline{\beta : \iota}.\, \overline{\tau} \to T\ \overline{\tau_u} \qquad \Gamma \vdash \overline{\tau_e : \iota} \qquad \Gamma \vdash \overline{e : \tau[\overline{\tau_e/\beta}]}}{\Gamma \vdash C\ \overline{\tau_u}\ \overline{\tau_e}\ \overline{e} : T\ \overline{\tau_u}}$$

6.5.4 Typing Unboxed Algebraic Data Types

In F_U^ω, data constructor type signatures have two additional components, a list of size parameter types and a list of static types, as discussed in Section 6.3.1. F_U^ω's typing rule for case expressions adds two groups of premises to the F^ω typing rule.

$$\frac{\Gamma \vdash C \geq (\overline{\tau_s})(\overline{\pi_s}) \Rightarrow \forall \overline{\beta : \iota}.\, \overline{\tau} \to T\ \overline{\pi} \qquad \Gamma \vdash e : T\ \overline{\pi} \qquad \Gamma, \overline{\beta : \iota}, \overline{x : \tau} \vdash c : \pi_c \qquad \Gamma \vdash \overline{d : \tau_s} \qquad \Gamma \vdash \overline{\pi_s \downarrow \varsigma}}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \overline{d} \Rightarrow C\ \overline{\beta}\ \overline{x}.\ c : \pi_c}$$

These premises relate the extra components of the signature $\overline{\tau_s}$ and $\overline{\pi_s}$ to the types used in the case expression. The given size parameters are matched to their expected types by $\Gamma \vdash \overline{d : \tau_s}$. The premises $\Gamma \vdash \overline{\pi_s \downarrow \varsigma}$ state that each static type has a statically known layout.

The typing rule for data expressions bears another difference from the fully boxed rule. In a fully-boxed formulation, a field of type τ is initialized from an expression of type τ. This is also true for boxed and value objects in F_U^ω. However, the value of a bare field of type τ is initialized by executing a parameter of type $\mathsf{Out}\ \tau \to \mathsf{Store}$. We define

$$\mathrm{init}(\kappa, \tau) = \begin{cases} \mathsf{Out}\ \tau \to \mathsf{Store} & \text{if } \kappa = \mathsf{bare} \\ \tau & \text{otherwise} \end{cases}$$

to stand for the data constructor parameter type corresponding to a field of type τ, where κ is the kind of τ. We also write $\Gamma \vdash e : \mathrm{init}(\tau)$ as shorthand for the two judgments $\Gamma \vdash \tau : \kappa$ and $\Gamma \vdash e : \mathrm{init}(\kappa, \tau)$. The typing rule for data expressions is shown below.

$$\frac{\begin{array}{c}\Gamma \vdash C \geq (\overline{\tau_s})(\overline{\pi_s}) \Rightarrow \forall \overline{\beta : \iota}.\, \overline{\tau} \to T\ \overline{\tau_u} \\ \Gamma \vdash \overline{\tau_e : \iota} \qquad \Gamma \vdash \overline{e : \mathrm{init}(\tau[\overline{\tau_e/\beta}])} \\ \Gamma \vdash \overline{d : \tau_s} \qquad \Gamma \vdash \overline{\pi_s \downarrow \varsigma}\end{array}}{\Gamma \vdash \overline{d} \Rightarrow C\ \overline{\tau_u}\ \overline{\tau_e}\ \overline{e} : \mathrm{init}(T\ \overline{\tau_u})}$$

The final two groups of premises are the same as in the rule for case expressions. The init function now appears in the typing rules for data constructor fields, indicating that initializers are used for bare fields. Likewise, init appears in the type of the data constructor expression, indicating that the data expression returns an initializer rather than a bare object.

6.6 Lowering from F_U^ω to $\lambda^{\mathbf{L}}$

After high-level optimizations are performed in F_U^ω, the compiler's job is to translate F_U^ω expressions to low-level primitive operations that can be straightforwardly implemented by machine instructions. The first step in this translation is to convert high-level F_U^ω to low-level λ^{L}. In the process, algebraic data structure operations are translated to memory accesses or manipulation of small, monomorphic products. Later steps of compilation generate C code to be compiled to object code.

The lowering algorithm for data and case expressions is presented in this seciton. We shall define the lowering translation $[\![e]\!]\,\Gamma$, which translates a F_U^ω expression e to an equivalent λ^{L} expression. Γ holds F_U^ω type information that guides the translation. Aside from the translation of data and case expressions, the lowering algorithm is trivial. Since λ^{L} is monomorphic, type abstraction and application disappear during lowering. Variables, integer literals, let expressions, function abstraction and application all translate directly from F_U^ω to λ^{L}. Variables and intermediate results of kind box, bare, and out are lowered to pointers, and their λ^{L} type is Ptr. Variables and intermediate results of kind val are lowered to λ^{L} values; their layout is their λ^{L} type. (Note that layouts ς and λ^{L} types σ have identical definitions.)

6.6.1 Computing the $\lambda^{\mathbf{L}}$ Structure of Objects

Each algebraic type definition is preprocessed to compute its low-level memory structure, that is, the object size and the offsets of an object's fields. Generally, the structure is a function of type parameters. For a type whose constructor signature is $\forall \overline{\alpha : \kappa}.\ (\overline{\tau_s})(\overline{\pi_s}) \Rightarrow \forall \overline{\beta : \iota}.\, \tau \to T\ \overline{\alpha}$, a structure-computing function is generated that takes an argument for each τ_s and each π_s and returns a product containing the offset of each field and the object size. For instance, the generated function for tuples is

$\mathsf{structureTuple} : \langle \mathsf{Int} \rangle \to \langle \mathsf{Int} \rangle \to \langle \mathsf{Int}, \mathsf{Int} \rangle \times \mathsf{Int}$
$\mathsf{structureTuple}\ s_1\ s_2 = \mathbf{let}\ \langle m \rangle = s_1\ \mathbf{in}\ \mathbf{let}\ \langle n \rangle = s_2\ \mathbf{in}\ \langle 0, m \rangle \times (m + n).$

The parameters are Sz objects. Since Sz objects are data structures with one field, they have type $\langle \mathsf{Int} \rangle$ after lowering and must be unpacked to get the size. The field offsets are computed by $\langle 0, m \rangle$, and the tuple size is computed by $m + n$.

Structure-computing functions are generated from data type definitions using a variant of the layout algorithm. The variant algorithm computes size values (that have type Int), rather than computing layouts. A size is computed for each field. Sizes are added together to produce field offsets and the object size. Structure-computing functions are saved in a lookup table indexed by data constructor names. We write structure(C) for the structure-computing function associated with data constructor C.

$$\text{read}(\Gamma, p, n, \tau) = \begin{cases} \mathbf{load}_\sigma\ (p+n) & \text{if } \Gamma \vdash \tau : \mathsf{val} \text{ and } \Gamma \vdash \tau \downarrow \sigma \\ \mathbf{load}_{\mathsf{Ptr}}\ (p+n) & \text{if } \Gamma \vdash \tau : \mathsf{box} \\ p+n & \text{if } \Gamma \vdash \tau : \mathsf{bare} \end{cases}$$

$$\text{write}(\Gamma, p, n, \tau, e) = \begin{cases} \mathbf{store}_\sigma\ (p+n)\ e & \text{if } \Gamma \vdash \tau : \mathsf{val} \text{ and } \Gamma \vdash \tau \downarrow \sigma \\ \mathbf{store}_{\mathsf{Ptr}}\ (p+n)\ e & \text{if } \Gamma \vdash \tau : \mathsf{box} \\ e\ (p+n) & \text{if } \Gamma \vdash \tau : \mathsf{bare} \end{cases}$$

Figure 6: Code generated for reading and writing object fields. Code generation uses a type environment Γ, object pointer p, field offset n, and field type τ.

6.6.2 Lowering Data and Case Expressions for Value Types

Algebraic value types are lowered to product types. A data expression simply constructs a product of its fields:

$$[\![C\ \overline{\tau_u}\ \overline{\tau_e}\ \overline{e}]\!]\,\Gamma = \langle\overline{[\![e]\!]\,\Gamma}\rangle$$

For instance, the unboxed tuple expression $\mathsf{tupleV}\ \mathsf{Int}\ \mathsf{Int}\ 0\ n$ is lowered to $\langle 0, n\rangle$.

A case expression unpacks a product into its component fields. The lowering algorithm also computes the types of variables introduced by the expression so that these types can be used when lowering the body of the expression.

$$[\![\mathbf{case}\ c\ \mathbf{of}\ C\ \overline{\beta}\ \overline{x}.\ e]\!]\,\Gamma = \mathbf{let}\ \langle\overline{x}\rangle = [\![c]\!]\,\Gamma\ \mathbf{in}\ [\![e]\!]\,(\Gamma, \overline{\beta : \iota}, \overline{x : \tau})$$
$$\text{where } \Gamma \vdash c : T\ \overline{\pi}$$
$$\Gamma \vdash C \geq (\overline{\tau_s})(\overline{\pi_s}) \Rightarrow \forall\overline{\beta : \iota}.\ \overline{\tau} \rightarrow T\ \overline{\pi}$$

The expression $\mathbf{case}\ t\ \mathbf{of}\ \mathsf{tupleV}\ x\ y.\ f\ x\ y$ is lowered to $\mathbf{let}\ \langle x, y\rangle = t\ \mathbf{in}\ f\ x\ y$. The pattern $\langle x, y\rangle$ unpacks t.

6.6.3 Lowering Data and Case Expressions for Boxed and Bare Types

Boxed and bare types reside in memory after lowering. A data expression on one of these types obtains a pointer to a new object, gets the address of each field, and then initializes the fields. A case expression reads the fields of an existing object.

Individual object fields are read and written differently depending on their storage strategy (Figure 6). We write $\text{read}(\Gamma, p, n, \tau)$ for reading an object of type τ from the address $p + n$, and $\text{write}(\Gamma, p, n, \tau, e)$ for using the object or initializer returned by e to write an object of type τ to the address $p + n$. For writing multiple fields, we extend read and write to lists: $\text{read}(\Gamma, p, \overline{n}, \overline{\tau})$ and $\text{write}(\Gamma, p, \overline{n}, \overline{\tau}, \overline{e})$. Boxed and value objects are read and written by transferring data between a variable and memory. The data is either an object or a pointer. Bare fields, on the other hand, cannot be transferred to a variable. Reading a bare field simply returns a pointer to the field. Writing a bare field is accomplished by passing the field's address to a given initializer function.

For example, to lower the data expression $\mathsf{stored}\ \mathsf{Int}\ 2$, we would first determine that the object being constructed has a field at offset 0 of type Int and kind val. The field is written by $\text{write}(\Gamma, p, 0, \mathsf{Int}, 2)$, which is the store operation $\mathbf{store}_{\mathsf{Int}}\ p\ 2$. The expression as a whole becomes the initializer function $\lambda p : \mathsf{Ptr}.\ \mathbf{store}_{\mathsf{Int}}\ p\ 2$.

In fact, the compiler does not directly generate this expression. It generates code to query the low-level structure of a $\mathsf{Stored}\ \mathsf{Int}$ object, then write its fields:

$\mathbf{let}\ \langle n\rangle \times m = \mathsf{structureStored}\ 1\ \mathbf{in}$
$\lambda p : \mathsf{Ptr}.\ \mathbf{store}_{\mathsf{Int}}\ (p + n)\ 2$

The function $\mathsf{structureStored}$ is a compiler-generated global function. Its argument, 1, is the size of a Int. The compiler will inline the call to $\mathsf{structureStored}$ and simplify it to produce $\lambda p : \mathsf{Ptr}.\ \mathbf{store}_{\mathsf{Int}}\ p\ 2$. Triolet's code generation strategy reflects a separation of concerns in the compiler. Layout computation is its own compiler module, separate from the lowering algorithm. A change to layout computation only changes how functions such as $\mathsf{structureStored}$ are generated. This makes it easier for us to implement more sophisticated layouts in Triolet.

The general algorithm for lowering data expressions on bare objects, of which Stored is a special case, is shown below.

$$\begin{aligned}
[\![\overline{d} \Rightarrow C\ \overline{\tau_u}\ \overline{\tau_e}\ \overline{e}]\!]\,\Gamma = {} & \mathbf{let}\ \langle\overline{n}\rangle \times m = \mathrm{structure}(C)\ \overline{[\![d]\!]\,\Gamma}\ \overline{\mathrm{size}(\varsigma)}\ \mathbf{in} \\
& \lambda p : \mathsf{Ptr}.\ \mathrm{write}(\Gamma, p, \overline{n}, \overline{\tau[\overline{\tau_e/\beta}]}, \overline{[\![e]\!]\,\Gamma}) \\
\text{where}\ & \Gamma \vdash C \geq (\overline{\tau_s})(\overline{\pi_s}) \Rightarrow \forall\overline{\beta : \iota}.\ \overline{\tau} \rightarrow T\ \overline{\tau_u} \\
& \Gamma \vdash \overline{\pi_s \downarrow \varsigma}
\end{aligned}$$

First, the low-level structure of a $T\ \overline{\tau_u}$ object is computed by calling the compiler-generated function structure(C). The function is passed run-time size information and statically computed sizes as arguments, and it returns field offsets $\overline{n}$ and an object size m. Then, an initializer function is created and returned. The initializer function takes an address, p, and writes each field at the appropriate offset from p.

Data expressions for boxed objects compute low-level structure and write fields in the same way. However, instead of creating an initializer, a new object is allocated and written to.

$$\begin{aligned}
& \mathbf{let}\ \langle\overline{n}\rangle \times m = \mathrm{structure}(C)\ \overline{[\![d]\!]\,\Gamma}\ \overline{\mathrm{size}(\varsigma)}\ \mathbf{in} \\
[\![\overline{d} \Rightarrow C\ \overline{\tau_u}\ \overline{\tau_e}\ \overline{e}]\!]\,\Gamma = {} & \mathbf{let}\ p = \mathbf{alloc}\ m\ \mathbf{in} \\
& \mathbf{let}\ \langle\rangle = \mathrm{write}(\Gamma, p, \overline{n}, \overline{\tau[\overline{\tau_e/\beta}]}, \overline{[\![e]\!]\,\Gamma})\ \mathbf{in}\ p \\
\text{where}\ & \Gamma \vdash C \geq (\overline{\tau_s})(\overline{\pi_s}) \Rightarrow \forall\overline{\beta : \iota}.\ \overline{\tau} \rightarrow T\ \overline{\tau_u} \\
& \Gamma \vdash \overline{\pi_s \downarrow \varsigma}
\end{aligned}$$

Case expressions for both boxed and bare objects are generated by lowering the scrutinee e, computing the low-level structure of the object being inspected, reading each field, and then lowering the body c of the case expression.

$$\begin{aligned}
& \mathbf{let}\ p = [\![e]\!]\,\Gamma\ \mathbf{in} \\
\left[\!\!\left[\begin{array}{l}\mathbf{case}\ e \\ \mathbf{of}\ \overline{d} \Rightarrow C\ \overline{\beta}\ \overline{x}.\ c\end{array}\right]\!\!\right]\Gamma = {} & \mathbf{let}\ \langle\overline{n}\rangle \times m = \mathrm{structure}(C)\ \overline{[\![d]\!]\,\Gamma}\ \overline{\mathrm{size}(\varsigma)}\ \mathbf{in} \\
& \mathbf{let}\ \langle\overline{x}\rangle = \mathrm{read}((\Gamma, \overline{\beta : \iota}), p, \overline{n}, \overline{\tau})\ \mathbf{in}\ [\![c]\!]\,(\Gamma, \overline{\beta : \iota}, \overline{x : \tau}) \\
\text{where}\ & \Gamma \vdash e : T\ \overline{\tau_u} \\
& \Gamma \vdash C \geq (\overline{\tau_s})(\overline{\pi_s}) \Rightarrow \forall\overline{\beta : \iota}.\ \overline{\tau} \rightarrow T\ \overline{\tau_u} \\
& \Gamma \vdash \overline{\pi_s \downarrow \varsigma}
\end{aligned}$$

7 From Fully Boxed Code to F_U^ω

To support unboxed polymorphism, F_U^ω introduces storage strategies, size parameters, restrictions on polymorphism, and initializers. These features make source code and the language semantics more complex than in a fully boxed language. Fortunately, it is not necessary to write directly in F_U^ω to get the benefits of F_U^ω. The Triolet compiler translates fully boxed code to F_U^ω as one of the first steps in compiling Triolet code.

The translation serves two vital purposes. First, it allows Triolet code to call library functions, defined in λ^{L}, that take or return unboxed data. Second, it minimizes the amount of pointer indirection in data structures by inserting pointer indirection only

when it is required for compatibility. To meet both these purposes, the translation generates code that boxes or unboxes objects: at function calls in the first case, and at case and data expressions in the second case.

On the other hand, the translation does not give Triolet code access to all the features of F_U^ω. Triolet code can't specify which storage strategies to use, and the translation does not attempt to optimize the storage strategies of temporary values, function parameters, or function results. Such low-level control is not too important, since conventional optimizations in F_U^ω are proficient at removing temporary objects and modifying parameter-passing conventions. Triolet code can be polymorphic over types of kind bare and box, but not val.

The compiler generates functions for computing Rep objects in a manner similar to how it computes object layouts (Section 7.2). These functions are used in the translation.

There are two steps to the translation. The first step occurs during type inference (Section 7.3). A Rep object is created for each object type that is not statically known, and these are used to supply size parameters of case and data expressions. The second step, which we call *elaboration* (Section 7.4), converts from a fully boxed type system to one with storage strategies. Adapter types and conversion code are inserted to make the converted code well-typed.

7.1 Introduction

To introduce the translation, we follow the steps of translation on the tupleApply function from Section 5.1. While the unboxed version of tupleApply from the introduction can be written in F_U^ω, Triolet's translation produces code that more closely conforms to the fully boxed calling convention. The fully boxed function definition is repeated here.

$$\mathsf{tupleApply} : \forall \alpha, \beta, \gamma : \star.\ (\alpha \to \beta \to \gamma) \to \mathsf{Tuple}\ \alpha\ \beta \to \gamma$$
$$\mathsf{tupleApply}\ \alpha\ \beta\ \gamma\ f\ t = \textbf{case}\ t\ \textbf{of}\ \mathsf{tuple}\ x\ y.\ f\ x\ y$$

First, we must give the function a valid F_U^ω type signature. We replace $\star$ by box. After making this change, the type Tuple α β is ill-kinded. To make the arguments properly kinded, we apply the type function AsBare to α and β.

$$\mathsf{tupleApply} : \forall \alpha, \beta, \gamma : \mathsf{box}.\ (\alpha \to \beta \to \gamma) \to \mathsf{Tuple}\ (\mathsf{AsBare}\ \alpha)\ (\mathsf{AsBare}\ \beta) \to \gamma$$

While this is now a valid type signature, the function body is likely to need run-time information about the type parameters (more precisely, about the bare types produced by applying AsBare to the type parameters). For each type parameter, we add a Rep parameter holding run-time type information.

$$\begin{aligned}\mathsf{tupleApply} : \forall \alpha, \beta, \gamma : \mathsf{box}.\ & \mathsf{Rep}\ (\mathsf{AsBare}\ \alpha) \to \mathsf{Rep}\ (\mathsf{AsBare}\ \beta) \to \mathsf{Rep}\ (\mathsf{AsBare}\ \gamma) \to \\ & (\alpha \to \beta \to \gamma) \to \mathsf{Tuple}\ (\mathsf{AsBare}\ \alpha)\ (\mathsf{AsBare}\ \beta) \to \gamma\end{aligned}$$

We insert several pieces of code to turn the fully-boxed function body into a valid F_U^ω expression. First, we insert code that computes the Rep object for a tuple, using the compiler-generated function repTuple. When the tuple's fields are read, they are bare objects (of types AsBare α and AsBare β), but the function f expects boxed arguments (of types α and β). To convert these objects, we insert calls to conversion functions. The functions copy and asBox extract conversion functions from r_α or r_β, and the extracted functions are immediately called. The resulting F_U^ω expression is shown below.

$$\begin{aligned}\mathsf{tupleApply} : \forall \alpha, \beta, \gamma : \mathsf{box}.\ & \mathsf{Rep}\ (\mathsf{AsBare}\ \alpha) \to \mathsf{Rep}\ (\mathsf{AsBare}\ \beta) \to \mathsf{Rep}\ (\mathsf{AsBare}\ \gamma) \to \\ & (\alpha \to \beta \to \gamma) \to \mathsf{Boxed}\ (\mathsf{Tuple}\ (\mathsf{AsBare}\ \alpha)\ (\mathsf{AsBare}\ \beta)) \to \gamma\end{aligned}$$

$\mathsf{tupleApply}\ \alpha\ \beta\ \gamma\ r_\alpha\ r_\beta\ r_\gamma\ f\ t =$
$\quad \textbf{let}\ r_t = \mathsf{repTuple}\ (\mathsf{AsBare}\ \alpha)\ (\mathsf{AsBare}\ \beta)\ r_\alpha\ r_\beta\ \textbf{in}$
$\quad \textbf{case}\ t\ \textbf{of}\ (\mathsf{size}\ (\mathsf{AsBare}\ \alpha)\ r_\alpha, \mathsf{size}\ (\mathsf{AsBare}\ \beta)\ r_\beta) \Rightarrow$
$\qquad\qquad \mathsf{tuple}\ x\ y.\ f\ (\mathsf{asBox}\ (\mathsf{AsBare}\ \alpha)\ r_\alpha\ (\mathsf{copy}\ (\mathsf{AsBare}\ \alpha)\ r_\alpha\ x))$
$\qquad\qquad\qquad (\mathsf{asBox}\ (\mathsf{AsBare}\ \beta)\ r_\beta\ (\mathsf{copy}\ (\mathsf{AsBare}\ \beta)\ r_\beta\ y))$

While this expression is considerably larger than the original, optimizations can eliminate the excess code. If tupleApply is inlined somewhere where the types α, β, and γ are constant, then the values of r_α, r_β, and r_γ will also be constant. Then, sizes can be evaluated statically and the uses of asBox and copy evaluate to direct calls that can be inlined and simplified.

We inserted unused Rep objects, just as the Triolet compiler does. Triolet inserts unused objects because type inference, which creates these objects, does not know which of them will be used during elaboration. In this case, r_α and r_β were indeed used, while r_γ and r_t were not. Unused objects are usually cleared away by dead code elimination.

7.2 Run-time type information

The Triolet compiler analyzes data type definitions to generate functions for creating Rep objects. These functions are used by the translation. For each value type constructor T, the analysis generates a function for computing its size as an object of type Sv $(T\ \overline{\tau})$. For each bare type construtor T, the analysis generates a function for computing an object of type Sz $(T\ \overline{\tau})$ and an object of type Rep $(T\ \overline{\tau})$. We name the compiler-generated functions sv$T$, sz$T$, and rep$T$, respectively.

Size-computing functions are generated similarly to Section 6.6.1. Field offsets are not computed. The process generates F_U^ω code instead of λ^{L} code and gives the code a type signature that reflects the size information being computed.

Two generic functions are used for boxing and unboxing bare objects. A boxed object is unboxed by copying its contents into a new bare object. The argument f is a copy function.

$\mathsf{genericAsBare} : \forall\alpha : \mathsf{bare}.\ \mathsf{Sz}\ \alpha \rightarrow (\alpha \rightarrow \mathsf{Out}\ \alpha \rightarrow \mathsf{Store}) \rightarrow \mathsf{AsBox}\ \alpha \rightarrow \mathsf{Out}\ \alpha \rightarrow \mathsf{Store}$
$\mathsf{genericAsBare}\ \alpha\ s\ f\ x\ y = \textbf{case}\ x\ \textbf{of}\ s \Rightarrow \mathsf{boxed}\ w.\ f\ w\ y$

A new bare object (given in the form of an initializer) is boxed by writing it into a new box.

$\mathsf{genericAsBox} : \forall\alpha : \mathsf{bare}.\ \mathsf{Sz}\ \alpha \rightarrow (\mathsf{Out}\ \alpha \rightarrow \mathsf{Store}) \rightarrow \mathsf{AsBox}\ \alpha$
$\mathsf{genericAsBox}\ \alpha\ s\ g = s \Rightarrow \mathsf{boxed}\ \alpha\ g$

We have cheated a bit with the type signatures; these functions treat a AsBox α as if it were a Boxed α, which is not true in general. In Triolet, evidence that AsBox α is equal to Boxed α is passed as an additional function parameter [35]. In particular, this equality does not hold when $\alpha = \mathsf{Ref}\ \tau$. The implementation for references is written manually.

A Rep contains a size and three functions. The size is computed by calling sz$T$. The first function copies an object by reading its fields and writing them to a new object. To determine how to copy fields, the data type definition is examined to find the fields' types and storage strategies. For all algebraic data types except references, the last two functions are simply instantiations of the generic conversion functions. For example, the following code is generated for Stored.

$\mathsf{szStored} : \forall\alpha : \mathsf{val}.\ \mathsf{Sv}\ \alpha \rightarrow \mathsf{Sz}\ (\mathsf{Stored}\ \alpha)$
$\mathsf{szStored}\ \alpha\ x = \textbf{case}\ x\ \textbf{of}\ \mathsf{sv}\ n.\ \mathsf{sz}\ (\mathsf{Stored}\ \alpha)\ n$

$$
\begin{array}{lrl}
\text{Kinds} & \kappa, \iota := & \mathsf{box} \mid \kappa \to \kappa \\
\text{Monotypes} & \tau, \pi := & \alpha \mid \tau\ \tau \mid \tau \to \tau \mid T \mid \mathsf{Int} \\
\text{Polytypes} & \sigma := & \forall \overline{\alpha : \kappa}.\ \overline{K\ \tau} \Rightarrow \tau \\
\text{Expressions} & c, d, e := & x \mid N \mid \mathbf{let}\ x = e\ \mathbf{in}\ e \mid e\ e \mid \lambda x.\ e \\
 & & \mid \mathbf{case}\ e\ \mathbf{of}\ C\ \overline{x}.\ e \mid C\ \overline{e}
\end{array}
$$

Figure 7: Syntax of the source language HM.

$$
\begin{array}{l}
\mathsf{repStored} : \forall \alpha : \mathsf{val}.\ \mathsf{Sv}\ \alpha \to \mathsf{Rep}\ (\mathsf{Stored}\ \alpha) \\
\mathsf{repStored}\ \alpha\ x = \mathbf{let}\ \mathsf{size} = \mathsf{szStored}\ \alpha\ x\ \mathbf{in} \\
\qquad \mathbf{let}\ \mathsf{copy} = \lambda y : \mathsf{Stored}\ \alpha.\ \lambda z : \mathsf{Out}\ (\mathsf{Stored}\ \alpha). \\
\qquad\qquad \mathbf{case}\ y\ \mathbf{of}\ \mathsf{stored}\ w.\ \mathsf{stored}\ \alpha\ w\ z\ \mathbf{in} \\
\qquad \mathbf{let}\ \mathsf{asBare} = \mathsf{genericAsBare}\ (\mathsf{Stored}\ \alpha)\ \mathsf{size}\ \mathsf{copy}\ \mathbf{in} \\
\qquad \mathbf{let}\ \mathsf{asBox} = \mathsf{genericAsBox}\ (\mathsf{Stored}\ \alpha)\ \mathsf{size}\ \mathbf{in} \\
\qquad \mathsf{rep}\ (\mathsf{Stored}\ \alpha)\ \mathsf{size}\ \mathsf{copy}\ \mathsf{asBare}\ \mathsf{asBox}
\end{array}
$$

References behaves specially with respect to boxing and unboxing because references are containers for boxed objects. Converting to bare form means writing a new reference. Converting to boxed form means reading a reference.

$$
\begin{array}{l}
\mathsf{repRef} : \forall \alpha : \mathsf{box}.\ \mathsf{Rep}\ (\mathsf{Ref}\ \alpha) \\
\mathsf{repRef}\ \alpha = \mathbf{let}\ \mathsf{size} = \mathsf{szRef}\ (\mathsf{Ref}\ \alpha)\ \mathbf{in} \\
\qquad \mathbf{let}\ \mathsf{copy} = \lambda y : \mathsf{Ref}\ \alpha.\ \lambda z : \mathsf{Out}\ (\mathsf{Ref}\ \alpha). \\
\qquad\qquad \mathbf{case}\ y\ \mathbf{of}\ \mathsf{ref}\ w.\ \mathsf{ref}\ \alpha\ w\ z\ \mathbf{in} \\
\qquad \mathbf{let}\ \mathsf{asBare} = \lambda y : \alpha.\ \lambda z : \mathsf{Out}\ (\mathsf{Ref}\ \alpha). \\
\qquad\qquad \mathsf{ref}\ \alpha\ y\ z\ \mathbf{in} \\
\qquad \mathbf{let}\ \mathsf{asBox} = \lambda y : \mathsf{Out}\ (\mathsf{Ref}\ \alpha) \to \mathsf{Store}. \\
\qquad\qquad \mathbf{case}\ \mathsf{size} \Rightarrow \mathsf{boxed}'\ (\mathsf{Ref}\ \alpha)\ w \\
\qquad\qquad \mathbf{of}\ \mathsf{size} \Rightarrow \mathsf{boxed}'\ y.\ \mathbf{case}\ y\ \mathbf{of}\ \mathsf{ref}\ z.\ z\ \mathbf{in} \\
\qquad \mathsf{rep}\ (\mathsf{Ref}\ \alpha)\ \mathsf{size}\ \mathsf{copy}\ \mathsf{asBare}\ \mathsf{asBox}
\end{array}
$$

We define prebuilt objects of type $\mathsf{Rep}\ (\mathsf{Stored}\ \tau)$ for zero-parameter value type constructors, for use by type inference. These objects are simply calls to $\mathsf{repStored}$ with appropriate arguments. For example, repInt is defined with type $\mathsf{Rep}\ (\mathsf{Stored}\ \mathsf{Int})$.

7.3 Type Inference

After parsing and eliminating syntactic sugar, Triolet code is in the implicitly typed form HM shown in Figure 7. This language corresponds to a subset of Haskell with strict evaluation semantics, or a subset of ML with higher kinds and type classes. Unlike F^ω_U, case and data expressions do not have size contexts, and all objects are boxed. As is the norm for Hindley-Milner typeable languages, types are stratified into monotypes τ and type schemes σ. Type schemes are quantified over type parameters and may contain class constraints $K\ \tau$, meaning that type τ is a member of class K. Fortunately for us, computing run-time type information is a lot like inferring type class membership [9]. It only takes a modest adjustment to a type class system [42] to compute run-time type information. The output of type inference is in a form quite similar to F^ω_U, but fully boxed and lacking some of F^ω_U's types. We call this intermediate language F^ω_I (Figure 8).

Type inference assigns data constructors a simplified type signature, of the form $\forall \overline{\alpha : \kappa}.\ \overline{\mathsf{Rep}\ \tau_s} \Rightarrow \overline{\tau} \to T\ \overline{\alpha}$. The class constraints $\overline{\mathsf{Rep}\ \tau_s}$ correspond to size parameter

Kinds	$\kappa, \iota := \mathsf{box} \mid \kappa \to \kappa$
Types	$\tau, \pi := \alpha \mid \tau\ \tau \mid \forall \alpha : \kappa.\, \tau \mid \tau \to \tau \mid T \mid \mathsf{Int}$

Figure 8: Syntax of kinds and types in F_I^ω. Syntax of expressions is identical to F_U^ω (Figure 3).

types in a F_U^ω type signature (Section 6.3.1). We have not implemented support for existential types in case and data expressions in HM.

7.3.1 Generating Type Class Instances

To generate Rep objects during type inference, we define a class of types that have run-time type information. The class is named Rep, and its dictionary type constructor is Rep. Type class instances are automatically generated from data type definitions. Instance declarations follow the same pattern as those described by Cheney et al. [9], albeit with different dictionary values. Each instance corresponds to a compiler-generated rep function; the parameters are the instance context, the return type is the instance head, and the function itself computes the instance dictionary. For example, repTuple has type $\forall \alpha, \beta : \mathsf{bare}.\ \mathsf{Rep}\ \alpha \to \mathsf{Rep}\ \beta \to \mathsf{Rep}\ (\mathsf{Tuple}\ \alpha\ \beta)$. The corresponding instance, written in Haskell syntax, is **instance** $(\mathsf{Rep}\ \alpha, \mathsf{Rep}\ \beta) \Rightarrow \mathsf{Rep}\ (\mathsf{Tuple}\ \alpha\ \beta)$. The instance declaration informs the type inference engine that, when it needs the representation of some tuple type $\mathsf{Tuple}\ \tau\ \pi$, it can derive it by calling repTuple on the representations of τ and π.

Although only bare types have run-time representations, type inference does not distinguish bare types from other types. Rep instances are generated for all proper types. An instances for a boxed type τ produces the representation of a $\mathsf{Ref}\ \tau$. An instance for a value type τ produces the representation of a $\mathsf{Stored}\ \tau$.

7.3.2 Generating Class Constraints and Size Parameters

To induce type inference to insert run-time representations into a program, Rep class constraints are generated in two ways during type inference. Elaboration may need run-time type information in order to insert conversions, such as the calls to copy and asBox in tupleApply. Type inference generates a constraint for the type of each expression. The value computed by the constraint is annotated onto the expression for use during elaboration. Constraints are also generated to fill in case and data expressions' size parameters. These constraints come from data constructor signatures.

Type inference produces the following code for tupleApply. The variables r_α, r_β, r_γ, and r_t are inserted, and variables are passed along to size contexts. Code for computing r_t is inserted based on the instance declaration for Tuple. Type inference also passes appropriate parameters to tupleApply anywhere that it is called.

$$\begin{aligned}
&\mathsf{tupleApply} : \forall \alpha, \beta, \gamma : \star.\ \mathsf{Rep}\ \alpha \to \mathsf{Rep}\ \beta \to \mathsf{Rep}\ \gamma \to (\alpha \to \beta \to \gamma) \to \mathsf{Tuple}\ \alpha\ \beta \to \gamma \\
&\mathsf{tupleApply}\ \alpha\ \beta\ \gamma\ r_\alpha\ r_\beta\ r_\gamma\ f\ t = \mathbf{let}\ r_t = \mathsf{repTuple}\ \alpha\ \beta\ r_\alpha\ r_\beta\ \mathbf{in} \\
&\qquad \mathbf{case}\ t\ \mathbf{of}\ (r_\alpha, r_\beta) \Rightarrow \mathsf{tuple}\ x\ y.\ f\ x\ y
\end{aligned}$$

7.4 Elaboration

After type inference, F_I^ω is elaborated to F_U^ω by introducing storage strategies into the fully boxed code. Elaboration uses the intermedate language F_E^ω, which is discussed

Kinds	$\kappa, \iota :=$ box | val | bare | init | $\kappa \to \kappa$
Types	$\tau, \pi := \alpha \mid \tau\ \tau \mid \forall \alpha : \kappa.\, \tau \mid \lambda \alpha : \kappa.\, \tau \mid \tau \to \tau$
	| T | Int | AsBare | AsBox | Init

Figure 9: Syntax of kinds and types in F^ω_E. Syntax of expressions is identical to F^ω_U (Figure 3).

below. The algorithm boils down to inserting the right conversions in the right places to generage well-typed output. When the translation produces kind mismatches in types, they are fixed by inserting adapter types. When the translation produces type mismatches in expressions, they are fixed by inserting coercion code. Our discussion of elaboration is organized in four parts. The first two parts deal with converting between F^ω_E types and translating types to F^ω_E. The next two parts deal with converting between F^ω_E expressions and translating expressions to F^ω_E.

It is convenient to regard initializers as a distinct storage strategy during elaboration. For this reason, elaboration produces F^ω_E code (Figure 9), which differs from F^ω_U only in the typing of initializers. Initializers are treated as members of an abstract data type Init τ of kind init. F^ω_E type environments are the same as F^ω_U type environments other than the difference in initializer types. At the end of elaboration, code is converted from F^ω_E to F^ω_U by adjusting the types of initializers: init is replaced by box, and Init τ is replaced by Out $\tau \to$ Store.

7.4.1 Canonical Types and Conversions on Types

To produce well-typed F^ω_U code, elaboration must select types in a globally consistent way. The translation associates each proper type in F^ω_I to at most one F^ω_U type of any given kind, called the *canonical form* of that kind. The notion of canonical forms comes from coercion-based unboxing transformations [36], where every type has one canonical boxed form. In Triolet, a given F^ω_I type can have multiple canonical forms, but at most one of a given kind. By translating types to their canonical forms, we ensure that a well-kinded translation is also well-typed. The canonical form of a type is wrapped in some combination of the adapter types Stored, Boxed, Ref, and Init. Since we disallow nesting of Boxed with Ref, there is only one well-kinded way to wrap a type with these adapters.

A type can be converted from one canonical form to another by applying or unapplying adapter types as shown in Figure 10. We write $(\kappa \triangleright \iota)$ for the conversion from kind κ to kind ι. Conversions on base kinds are defined so that any round-trip conversion leaves its argument unchanged; for instance, $(\mathsf{bare} \triangleright \mathsf{box})\,((\mathsf{box} \triangleright \mathsf{bare})\,\tau) = \tau$. A conversion on arrow kinds becomes a type operator that converts its domain and range.

7.4.2 Type Elaboration

Type elaboration brings each F^ω_I type to a particular canonical form that we call the *natural form*. The natural form of a type is the one that is not wrapped in adapters. An F^ω_I type may be translated to a desired canonical form of kind κ by producing the natural form, then converting it to kind κ. The type elaboration algorithm is shown in Figure 11. The natural form of type τ is $\mathcal{N}[\![\tau]\!]\Gamma$, and its canonical form of kind κ is $\mathcal{C}_\kappa[\![\tau]\!]\Gamma$. The F^ω_E type environment Γ is used for computing kinds. To lighten the notation, Γ is implicitly passed as the type environment; for instance, $\mathcal{N}[\![\tau]\!]$ is

$$(\mathsf{val} \triangleright \mathsf{bare})\,\tau = \mathsf{Stored}\ \tau \qquad (\mathsf{bare} \triangleright \mathsf{val})\ (\mathsf{Stored}\ \tau) = \tau$$
$$(\mathsf{bare} \triangleright \mathsf{init})\,\tau = \mathsf{Init}\ \tau \qquad (\mathsf{init} \triangleright \mathsf{bare})\ (\mathsf{Init}\ \tau) = \tau$$
$$(\mathsf{bare} \triangleright \mathsf{box})\,\tau = \mathsf{AsBox}\ \tau \qquad (\mathsf{box} \triangleright \mathsf{bare})\,\tau = \mathsf{AsBare}\ \tau$$

$$(\kappa \triangleright \iota)\,\tau = (\mathsf{bare} \triangleright \iota)\,((\kappa \triangleright \mathsf{bare})\,\tau)$$
$$(\kappa \triangleright \kappa)\,\tau = \tau$$
$$(\kappa \to \iota \triangleright \kappa' \to \iota')\,\tau = \lambda\alpha : \kappa'.\ (\iota \triangleright \iota')\left(\tau\ ((\kappa' \triangleright \kappa)\,\alpha)\right)$$

Figure 10: Kind conversions. $(\kappa \triangleright \iota)$ is a conversion from κ to ι.

$$\begin{aligned}
\mathcal{N}[\![\alpha]\!] &= \alpha \\
\mathcal{N}[\![T]\!] &= T \\
\mathcal{N}[\![\mathsf{Int}]\!] &= \mathsf{Int} \\
\mathcal{N}[\![\tau\ \pi]\!] &= \mathcal{N}[\![\tau]\!]\ \mathcal{C}_\kappa[\![\pi]\!] \\
&\quad \text{where } \Gamma \vdash \mathcal{N}[\![\tau]\!] : \kappa \to \iota \\
\mathcal{N}[\![\forall\alpha : \kappa.\ \tau]\!] &= \forall\alpha : \kappa.\ \mathcal{C}_{\mathsf{box}}[\![\tau]\!](\Gamma, \alpha : \kappa) \\
&\quad \text{where } \Gamma \vdash \mathcal{N}[\![\tau]\!] : \iota \\
\mathcal{N}[\![\tau \to \pi]\!] &= \mathcal{C}_{\mathsf{box}}[\![\tau]\!] \to \mathcal{C}_{\mathsf{box}}[\![\pi]\!] \\
\mathcal{C}_\kappa[\![\tau]\!] &= (\iota \triangleright \kappa)\,\mathcal{N}[\![\tau]\!] \\
&\quad \text{where } \Gamma \vdash \mathcal{N}[\![\tau]\!] : \iota
\end{aligned}$$

Figure 11: Type elaboration.

shorthand for $\mathcal{N}[\![\tau]\!]\Gamma$. When the environment is extended, the extended environment is passed explicitly.

Canonical forms are called for in three places in the algorithm. The important one is type application, where the argument is converted to the kind expected by the operator. This conversion ensures that polymorphic fields are unboxed when possible. Canonical boxed forms are also called for in $\to$ and $\forall$ types. We require canonical forms to be unique, but $\to$ and $\forall$ allow subterms to have different kinds, so we force these subterms to be boxed.

7.4.3 Type Coercions

When elaboration encounters a kind mismatch between values, it inserts coercion code. A coercion converts an expression's result to a canonical form of the right kind. The coercions used by elaboration are listed in Figure 12. Related conversions are grouped into pairs. The first two rows convert to and from val using the stored data constructor. The next two rows convert to and from box using the conversion functions taken from a Rep object. The final two rows convert between bare and init by copying data or by executing an initializer. We write coerce(Γ, τ, π, e) for the expression that evaluates e to a result with canonical type τ, then coerces it to canonical type π. The F_E^ω type environment Γ is used for computing kinds. An object can be coerced between any two canonical forms using coerce.

There is more to the story, because not all F_E^ω objects have a type that is canonical. Canonical function types take boxed parameters and return boxed results; however,

From	To	Type	Expression	Run-time effect
val	init	$\tau \to \mathsf{Init}\ (\mathsf{Stored}\ \tau)$	$\mathsf{stored}\ \tau\ e$	Store a value
bare	val	$\mathsf{Stored}\ \tau \to \tau$	$\mathbf{case}\ e\ \mathbf{of}\ \mathsf{stored}\ x.\ x$	Load a value
box	init	$\mathsf{AsBox}\ \tau \to \mathsf{Init}\ \tau$	$\mathsf{asBare}\ \tau\ d\ e$	Copy an object or store a pointer
init	box	$\mathsf{Init}\ \tau \to \mathsf{AsBox}\ \tau$	$\mathsf{asBox}\ \tau\ d\ e$	Allocate memory and possibly load a pointer
bare	init	$\tau \to \mathsf{Init}\ \tau$	$\mathsf{copy}\ \tau\ d\ e$	Copy an object
init	bare	$\mathsf{Init}\ \tau \to \tau$	$\mathbf{let}\ s = \mathsf{size}\ \tau\ d\ \mathbf{in}$ $\mathbf{case}\ s \Rightarrow \mathsf{boxed}'\ \tau\ e$ $\mathbf{of}\ s \Rightarrow \mathsf{boxed}'\ x.\ x$	Allocate memory

Figure 12: Coercions between canonical types. Each row shows the code that is inserted to coerce the result of e from one canonical form to another. Where d appears, it is an expression of type $\mathsf{Rep}\ \tau$ computed by type inference.

$$(\forall\alpha:\kappa.\,\tau \blacktriangleright \forall\beta:\iota.\,\pi)_\Gamma\, e = \Lambda\beta:\iota.\ (\tau[(\iota \triangleright \kappa)\,\beta/\alpha] \blacktriangleright \pi)_{(\Gamma,\beta:\iota)}\,(e\ ((\iota \triangleright \kappa)\,\beta))$$
$$(\tau \to \pi \blacktriangleright \tau' \to \pi')_\Gamma\, e = \lambda x:\tau'.\ (\pi' \blacktriangleright \pi)_\Gamma\,\big(e\ ((\tau' \blacktriangleright \tau)_\Gamma\, x)\big)$$
$$(\tau \blacktriangleright \pi)_\Gamma\, e = \mathrm{coerce}(\Gamma,\tau,\pi,e)$$

Figure 13: Coercion between non-canonical types. $(\tau \blacktriangleright \pi)_\Gamma$ is a coercion from τ to π with respect to environment Γ.

in order to reduce the amount of boxing and unboxing associated with function calls, functions are allowed to take and return unboxed data. To deal with non-canonical functions, we extend coercions to non-canonical $\to$ and $\forall$ types as shown in Figure 13. A coercion on a function is is pushed under a lambda abstraction in the usual way for coercion-based unboxing transformations. A coercion on a type abstraction is similar.

7.4.4 Expression Elaboration

F^ω_I expresions are translated to F^ω_E expressions by the function $\mathcal{E}$ in Figure 14. Subexpressions are elaborated with $\mathcal{E}$ and type expressions with $\mathcal{N}$, then types are converted or expressions coerced as demanded by the type of the elaborated code. As before, $\mathcal{E}[\![e]\!]$ is shorthand for $\mathcal{E}[\![e]\!]\Gamma$. We write subscripted $\mathcal{E}$ functions for elaboration followed by inserting a coercion; subscripts will be explained as they come up. Variable and literal expressions do not need coercion. We discuss the other coercions in turn below.

In a let expression, the right-hand side, c, should be executed no more than once. However, if c were allowed to return an initializer, the initializer could be executed each time that x is used. That is, elaboration could produce an expression like $\mathbf{let}\ x = (\lambda y{:}\mathsf{Out}\ \alpha.\mathsf{expensive}\ 10\ y)\ \mathbf{in}\ f\ (s \Rightarrow \mathsf{boxed}\ \alpha\ x)\ (s \Rightarrow \mathsf{boxed}\ \alpha\ x)$, causing $\mathsf{expensive}\ 10\ y$ to be executed twice instead of once. To prevent this from happening, the elaboration algorithm coerces initializer types to bare types. Other types are not coerced. We write $\mathcal{E}_{\mathrm{use}}[\![c]\!]$ for elaboration to a non-initializer form.

In type application and function application, the operator e is first coerced into its natural form, removing any adapter types from it. We write $\mathcal{E}_\mathcal{N}$ for elaboration to natural form. Based on the operator's type, the argument τ of a type application is coerced to the expected kind by $\mathcal{C}_\kappa$, or the argument c of an application is coerced to

$$
\begin{aligned}
&\mathcal{E}[\![x]\!] &&= x\\
&\mathcal{E}[\![N]\!] &&= N\\
&\mathcal{E}[\![\mathbf{let}\ x = c\ \mathbf{in}\ e]\!]\Gamma &&= \mathbf{let}\ x = \mathcal{E}_{\text{use}}[\![c]\!]\ \mathbf{in}\ \mathcal{E}[\![e]\!](\Gamma, x : \tau)\\
& &&\quad \text{where } \Gamma \vdash \mathcal{E}_{\text{use}}[\![c]\!] : \tau\\
&\mathcal{E}[\![e\ \tau]\!] &&= \mathcal{E}_{\mathcal{N}}[\![e]\!]\ \mathcal{C}_{\kappa}[\![\tau]\!]\\
& &&\quad \text{where } \Gamma \vdash \mathcal{E}_{\mathcal{N}}[\![e]\!] : \forall\alpha : \kappa.\, \pi\\
&\mathcal{E}[\![e\ c]\!] &&= \mathcal{E}_{\mathcal{N}}[\![e]\!]\ \mathcal{E}_{\tau}[\![c]\!]\\
& &&\quad \text{where } \Gamma \vdash \mathcal{E}_{\mathcal{N}}[\![e]\!] : \tau \to \pi\\
&\mathcal{E}[\![\Lambda\alpha : \kappa.\, e]\!] &&= \Lambda\alpha : \kappa.\, \mathcal{E}[\![e]\!](\Gamma, \alpha : \kappa)\\
&\mathcal{E}[\![\lambda x : \tau.\, e]\!] &&= \lambda x : \mathcal{N}[\![\tau]\!].\, \mathcal{E}[\![e]\!](\Gamma, x : \mathcal{N}[\![\tau]\!])\\
&\mathcal{E}[\![\overline{d} \Rightarrow C\ \overline{\tau}\ \overline{\pi}\ \overline{e}]\!] &&= \overline{\mathcal{E}_{\text{size}}[\![d]\!]} \Rightarrow C\ \overline{\mathcal{C}_{\kappa}[\![\tau]\!]}\ \overline{\mathcal{C}_{\iota}[\![\pi]\!]}\ \overline{\mathcal{E}_{\tau_f}[\![e]\!]}\\
& &&\quad \text{where } \Gamma \vdash C \geq (\overline{\tau_s})(\overline{\pi_s}) \Rightarrow \forall\overline{\beta : \iota}.\, \overline{\tau_c} \to T\ \overline{\mathcal{C}_{\kappa}[\![\tau]\!]}\\
& &&\quad\quad \overline{\tau_f = \text{init}(\tau_c[\overline{\mathcal{C}_{\iota}[\![\pi]\!]/\beta}])}\\
&\mathcal{E}[\![\mathbf{case}\ c\ \mathbf{of}\ \overline{d} \Rightarrow C\ \overline{\beta}\ \overline{x}.\, e]\!] &&= \mathbf{case}\ \mathcal{E}_{\mathcal{N}}[\![c]\!]\ \mathbf{of}\ \overline{\mathcal{E}_{\text{size}}[\![d]\!]} \Rightarrow C\ \overline{\beta}\ \overline{x}.\, \mathcal{E}[\![e]\!](\Gamma, \overline{\beta : \kappa}, \overline{x : \tau_c})\\
& &&\quad \text{where } \Gamma \vdash C \geq (\overline{\tau_s})(\overline{\pi_s}) \Rightarrow \forall\overline{\beta : \iota}.\, \overline{\tau_c} \to T\ \overline{\tau}\\
& &&\quad\quad \Gamma \vdash \mathcal{E}_{\mathcal{N}}[\![c]\!] : T\ \overline{\tau}
\end{aligned}
$$

Figure 14: Expression elaboration.

the expected type by $\mathcal{E}_{\tau}$.

Type abstraction is elaborated simply by elaborating the body of the abstraction. Function abstraction is elaborated similarly, except that we also convert the function argument to its natural form.

When elaborating a data expression, the data constructor's F_E^{ω} signature determines how to elaborate the given arguments. The given type parameters $\overline{\tau}$ and $\overline{\pi}$ are translated to canonical forms of kinds κ and ι taken from the data constructor's signature. Using the elaborated type parameters, the field types $\overline{\tau_f}$ are computed. Finally, the given field expressions $\overline{e}$ are elaborated and coerced. We write $\mathcal{E}_{\text{size}}$ to elaborate an expression, which must evaluate to a Rep object, and extract the size from the object. If $\mathcal{E}[\![d]\!]$ produces expression d' of type Rep τ, then $\mathcal{E}_{\text{size}}[\![d]\!]$ produces the expression size $\tau\ d'$.

A case expression again uses the data constructor's F_E^{ω} signature, this time to determine the types of the bound variables. First, the scrutinee c is elaborated and coerced to its natural form. The scrutinee's type determines the field types τ_c. The case-bound pattern variables are added to the type environment when elaborating the body e.

7.5 Backward Translation from F_U^{ω} to HM

Type inference and elaboration are guided by type signatures of type constructors, data constructors, and library functions. These signatures are translated from F_U^{ω} back to HM and F_E^{ω}. Since HM's type system is less flexible than F_U^{ω}'s, only a subset of types can be represented. Type signatures that cannot be translated are removed from the type environment, making them invisible to Triolet code but still usable internally. We summarize the translation steps here.

The translation from HM to F_E^{ω} inserts initializer types. Any type of the form Out $\tau \to$ Store is replaced by Init τ. Types that still mention Out or Store after this

replacement are removed from the environment.

The translation from F_E^ω to F_I^ω eliminates adapter types, polymorphism over kind val, and storage strategies. In type signatures, adapter types AsBare, AsBox, Boxed, Ref, Stored, and Init are removed, leaving their argument in place. Types are removed from the environment if they mention type variables of kind val. In constructor signatures, size parameter types of the form Sz τ are replaced by Rep τ. When translating from F_I^ω to HM, types that have the same form as a type scheme are converted to type schemes, while others are removed.

8 Implementation Status

This paper documents our current implementation of the Triolet compler, but also amends our original design in ways that have not yet been put into practice. While the differences are more or less invisible to an end user, the amended design makes the compiler's internals more robust and extensible. We discuss the major differences, and how our experience has led us to change the design, below.

Size parameters are not yet a part of the Triolet compiler. Instead, the lowering algorithm performs a local analysis to find variables holding run-time size information. While we have made this design work, it is unsatisfactory because it introduces irritating implicit dependences into the compiler that sometimes manifest as compile-time errors. A variable that appears to be unused may actually be needed by the lowering algorithm, so the optimizer must be cautious when removing code. An apparently valid F_U^ω library function may be missing a parameter, causing lowering to fail because it cannot find information that it needs. Size parameters make these dependences explicit, helping to explain why the compiler works and closing off a class of potential bugs.

Similarly, static type parameters and restrictions on value polymorphism are not currently in the Triolet compiler. The compiler currently expects not to see a type variable of kind val in certain situations. Our additions to the type system justify these expectations and translate them into meaningful requirements. Elaboration does not generate code that violates these assumptions, but manually written F_U^ω code can.

In the current compiler, various code generation modules and handwritten functions make implicit assumptions about memory layout. The low-level structure of an object is computed in the lowering algorithm. Functions that compute information about algebraic data types (such as the functions szTuple and repTuple) are manually written for each type. The difficulty of making globally consistent memory layout assumptions prompted us to consider how to derive all layout-related code from a single memory layout specification. Tthe led to the design described in this paper.

The data types used for run-time representations (Rep, Sz, etc.) are implemented as we have described. Type inference is the same, except that it does not insert size parameters or annotate expressions with run-time representations. The differences in type inference entail some differences in elaboration, but it is largely the same as we have described.

9 Evalution

To evaluate the impact of unboxed polymorphism on real algorithms in the context of an optimizing compiler, we compile a set of benchmarks that represent our target use cases and measure their performance and memory allocation characteristics. These benchmarks embody numerical algorithms that are written in terms of polymorphic, higher-order library functions that read and/or write arrays.

For each benchmark, we compare three versions of the benchmark's most computationally demanding loops: C, unboxed, and boxed. We seek the answers to two questions: how closely does Triolet code approach the performance of C code, and how much does array unboxing contribute to that performance? The C version consists of handwritten, monomorphic code, representing the best possible outcome of unboxing optimizations. All storage is preallocated. Small values are unboxed in local variables. The remainder of the data consists of a few large arrays containing integer and floating-point values. The unboxed version is Triolet code with our unboxing optimization. The boxed version is the same Triolet code, except that array elements are boxed. Both Triolet versions undergo the same compiler optimizations and unbox the same fields of other data structures.

In experiments, C code was compiled with `gcc -O3`. Triolet uses `gcc -O2` as its backend compiler. All experiments were run on one core of a 2.66 GHz Intel Core 2 Quad processor with a 4 MiB last-level cache.

Data sets were sized to fit in the cache so that cache miss latency does not obscure differences between the C and unboxed benchmark versions. The C and unboxed versions have the same data layout and access patterns (with exceptions discussed in Section 9.1), so they would have the same cache miss behavior as well. Only the time and allocation spent in the execution of numerical loops was measured. Measurements are the average of ten consecutive runs.

Triolet uses an off-the-shelf, conservative garbage collector (GC) [3]. It is likely that a collector tuned for the memory usage characteristics of functional code could achieve better performance. To estimate the overhead due to GC, we also measure "no-GC" performance with GC disabled. In no-GC measurements, memory is bump-allocated from a 1 GiB heap and freed between runs of each loop.

9.1 Benchmarks

Four of our benchmarks (`cutcp`, `mri-q`, `sgemm`, and `tpacf`) are taken from the Parboil benchmark suite [38]. We use Parboil's "base" version as our C version. The Parboil benchmarks contain numerical algorithms drawn from a variety of applications. We selected four benchmarks that use nested loops to construct lists, reduce values, or scatter values. We have not ported and evaluated the rest of the Parboil suite; a complete evaluation is left to future work. We wrote two additional benchmarks (`sobel` and `smvm`) to represent common numerical algorithms that are not in Parboil. Sobel maps a 3×3 stencil function over an input image to produce an output image. Smvm multiplies a matrix in compressed sparse row format by a vector.

To demonstrate that Triolet can generate polymorphic code, we wrote two variants of a microbenchmark (`power-i` and `power-r`) that computes the powerset of a list iteratively or recursively. The iterative variant is an efficient implementation that selects subsets of an array based on the set bits in a counter. The recursive variant, which is only implemented in Triolet, is a simple algorithm used for pedagogical purposes [1] that uses a loop to construct an iterator over the powerset of a list. The Nth loop iteration constructs an iterator over the powerset of a sublist of length N. Because the structure of the computed iterator is input-dependent, it cannot be simplified to a loop. Moreover, because the iterator's state is also input-dependent, this benchmark requires support for polymorphic code generation.

To isolate the influence of unboxing from other optimizations, we have written the Triolet code to use the same loop structure and data structures as the C code whenever possible. We discuss discrepancies and their performance effects below. Loop optimizations (such as tiling and unrolling) are not employed in any benchmark version. The loop structure of `cutcp` is different in C and Triolet. The C version of `cutcp` sorts inputs for better cache locality while the Triolet version does not. The Triolet version

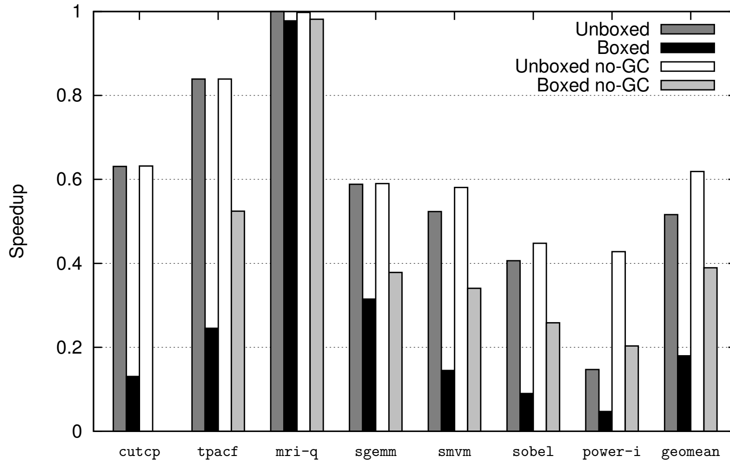

Figure 15: Execution time of Triolet benchmarks relative to C.

of `mri-q` fuses two loops that are separate in C. Triolet currently supports only 4-byte integers, but `tpacf` and `sobel` use other sizes. For all these benchmarks, changing the C version to match the Triolet version produced at most a 1% speedup or 4% slowdown, much smaller than the overall performance difference between C and Triolet. The C version of `power-i` precomputes the total number of outputs and writes them to a flat array. The Triolet version generates an array of arbitrary-length arrays. The Triolet code does not explicitly compute array sizes, relying instead on the implementation to size arrays properly. Changing the C version to generate an array of arbitrary-length arrays slowed it down by a factor of 2.95× due to the additional memory allocation and copying.

9.2 Execution Time

Benchmark execution time is compared in Figure 15. The height of a bar is its speedup relative to C. Since we are comparing similar algorithms, we can think of the C version as a measure of the useful work the algorithm performs, so that the height of a bar is the fraction of execution time spent on useful work and the gap above a bar is the fraction lost to overhead. Unboxed and boxed run at (respectively) 52% and 18% the speed of C on average.

The 2.87× speedup in going from boxed to unboxed arrays comes from a combination of reduced allocator overhead, a reduced memory footprint, and less manipulation of pointer values. Unboxing arrays reduces memory consumption by eliminating pointers to array elements and eliminating boxed object headers on array elements. To estimate allocator overhead, we measure performance using a simple bump allocator. This allocator does not automatically reclaim memory, making it very fast but impractical for real applications. With bump allocation, unboxing arrays produces a smaller 1.59× speedup (excluding `cutcp`, which runs out of memory with boxed arrays).

Unboxing also yields a 1.24× in the polymorphic code generated from `power-r`, though this speedup is dwarfed by iterator overhead. Unboxed `power-r` runs at 1.5% and boxed `power-r` at 3.7% the speed of the corresponding `power-i` version. The no-GC versions, while faster, are still much slower than `power-i`.

Benchmark	Bytes of output	Bytes allocated/bytes output		Objects allocated	
		Unboxed	Boxed	Unboxed	Boxed
`cutcp`	2.10 Mi	1.00	1.4×10^3	17	256 M
`tpacf`	16.5 Ki	2.63	2.5×10^4	1.51 k	35.7 M
`mri-q`	311 Ki	1.00	2.84	16	35.8 k
`sgemm`	80.0 Ki	1.00	5.00	18	20.5 k
`smvm`	17.2 Ki	1.01	5.01	6	4.41 k
`sobel`	1.00 Mi	1.00	5.00	24	262 k
`power-i`	44 Ki	3.24	8.85	3.61 k	16.9 k

Figure 16: Memory allocation of Triolet benchmarks.

9.3 Memory Allocation

We can observe the effects of unboxing more directly by examining the benchmarks' memory allocation behavior in Figure 16. The first column shows the number of bytes of numeric values written to arrays by Triolet code. This is (modulo the aforementioned differences in integer sizes) also the number of bytes written by C code.

The next two columns show the number of bytes actually allocated per byte of numeric data. Only two of the unboxed benchmarks allocate more than 1% extra memory. In contrast, the most allocation-efficient boxed benchmark allocates 1.84 extra bytes per byte, and the least efficient one allocates over ten kilobytes of memory per byte of output! Ideally, the unboxed column would be filled with ones, but two of these benchmarks allocate significant extra data. In `tpacf`, a single call to Triolet produces only 80 bytes of data, which is not enough to amortize the overhead of creating object headers, array metadata, and `Rep` objects. The extra allocation in `power-i` is primarily due to array resizing. Values are conditionally appended to arrays, reallocating when an array runs out of capacity. In contrast, the C version precomputes the total output size.

The dramatic amount of allocation in boxed `cutcp` and `tpacf` is due to the allocation of short-lived integer or floating-point values. These benchmarks compute a large number of independent sums by repeatedly updating an array in-place. In the unboxed versions, an integer or floating-point value is read and overwritten without allocating memory. In the boxed versions, arrays of pointers are read and overwritten. The allocation overhead is due to the creation of a new boxed number on each array update. In `sgemm`, `smvm`, and `sobel`, each boxed 4-byte number occupies 20 bytes of space (leading to the observed overhead factor of 5): the number itself, an 8-byte boxed object header, and an 8-byte pointer to the object. The factor is smaller in `mri-q` because it operates on arrays of tuples, whose fields are unboxed.

Finally, by counting the number of allocated objects, we can verify that most unboxed benchmarks create only a handful of objects. There are 201 calls to Triolet code in `tpacf`, allocating about 7.5 objects per call. Only unboxed `power-i` creates many objects, because it outputs many small arrays.

10 Related Work

The run-time costs of boxing have been tackled from several angles.

Unboxed types. Unboxing and polymorphism are tough to combine. Some languages let users choose between the two: boxed types may be used in a polymorphic context, while value types are unboxed and must not be used in a polymorphic context. To smooth over the incompatibility between boxed and value types, a language may

implicitly convert between corresponding boxed and value types [17], or provide boxed wrappers around value types [31].

Eliminating polymorphism. Some languages are monomorphized during compilation by creating a monomorphic copy of each polymorphic function for each type at which it is used. However, this strategy cannot be guaranteed to succeed in languages with polymorphic recursion, existential types, or first-class parametric polymorphism [26].

A language implementation can reconcile monomorphization with advanced type system features by compiling code dynamically. At run time, the implementation detects which unboxed types are actually used by a program and generates monomorphic code for only those types. This can be done by just-in-time compiling a language [21] or by embedding a monomorphizable language into another language [33]. Such implementations rely on heavyweight run-time support to compute type information, compile functions, and cache compiled code.

Coercion-based unboxing. Coercion-based unboxing transformations use statically available type information to transform a fully boxed program into one that manipulates a mixture of boxed and unboxed data [24, 41, 36]. Data is represented in unboxed form when its type is statically known, and boxed otherwise; systems vary in precisely what they unbox. When data is passed from a polymorphic context to a monomorphic one, or vice versa, it is *coerced* between its boxed and unboxed representations by rebuilding the data in the other representation. Because coercions carry a run-time cost proportional to the size of the data being coerced, only small, nonrecursive objects are unboxed in practice.

Although Triolet's elaboration stage is a coercion-based transformation, its purpose is only to minimize the use of boxing inside data structures, rather than globally. Triolet relies instead on local optimizations to eliminate unnecessary boxing in other places. Prior empirical results indicate that local optimizations can confer most of the benefit of coercion-based unboxing [25].

Object inlining is an object-oriented program optimization similar to unboxing [13]. Because it is used on mutable objects, coercions (which build a copy of an object) cannot be used. Instead, object inlining transforms object layouts globally.

Type-level computation. Objects can be more aggressively represented in unboxed form if some type information is passed around at run time. Whereas coercion-based systems convert data into the representation that a function is able to access, type-passing functions accept various data representations and decide how to access data by inspecting its type. Type-passing code can represent data in unboxed form without ever coercing it to boxed form, avoiding the overhead of coercion. On the other hand, the need for run-time dispatch can make object access in polymorphic code slower.

Tagging is simple example of this approach, and it has been used for unboxing objects in polymorphic code [27, 25]. More general approaches employ type-level computation, that is, they compute and inspect types at run time. Types can be functions of other types [18, 35]. Run-time dispatch is effected by inspecting types [18], proxies for types [11], or type class dictionaries [6].

Type-level computation has been proposed as a way to unbox data types. Several works use tagging or type-level computation to unbox arrays [26, 25, 4, 20]. Regardless of the type system's expressiveness, however, these methods are confined by the underlying language implementation's data layout strategies. For instance, none of these methods can unbox an $\mathsf{Arr}\ 100\ (\mathsf{Maybe}\ (\mathsf{Ref}\ (\mathsf{Int} \rightarrow \mathsf{Int})))$ into one block of memory with 100 tags and 100 pointers, because that memory layout is not supported. This type can be

vectorized [4] with unboxed arrays, but a vectorized array is not an array: for instance, it does not support $\Theta(1)$ random-access mutation. Our work complements type-level computation by giving more data layout flexibility to the language implementation.

11 Conclusion

In numerical functional programs written using high-level polymorphic library functions, the efficiency of data access in polymorphic code has a large impact on overall execution time. We have demonstrated that functional numerical algorithms can compile to code that operates on unboxed arrays and executes at 52% of the speed, on average, of the same algorithms written in monomorphic C. This is 2.87× faster than if boxed arrays were used. To support unboxing in polymorphic code, we described a type-safe extension to System F^{ω} and a translation from fully boxed code to this extension. Unlike previous approaches, our method can be used at compile time in the presence of first-class polymorphic values and is not limited to unboxing a preselected set of types.

Acknowledgments

The work is partly supported by the FCRP Giga Scale Research Center, the DoE Vancouver Project (DE-FC02-10ER26004/DE-SC0005515), the Intel/Microsoft Illinois UPCRC, and the UIUC CUDA Center of Excellence.

References

[1] J. Anderson, R. Farrell, and R. Sauers. Learning to program in LISP. *Cognitive Science*, 8(2):87–129, 1984.

[2] G. Blelloch, J. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.

[3] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software—Practice & Experience*, 18(9):807–820, 1988.

[4] M. Chakravarty and G. Keller. More types for nested data parallel programming. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, pages 94–105, 2000.

[5] M. Chakravarty and G. Keller. An approach to fast arrays in Haskell. In *Lecture Notes for the 2002 Summer School and Workshop on Advanced Functional Programming*, pages 27–58, 2003. LNCS 2638.

[6] M. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *Proc. ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 1–13, 2005.

[7] M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A status report. In *Proc. Workshop on Declarative Aspects of Multicore Programming*, pages 10–18, 2007.

[8] A. Charguéraud and F. Pottier. Functional translation of a calculus of capabilities. In *Proc. ACM SIGPLAN international conference on Functional programming*, pages 213–224, 2008.

[9] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Proc. ACM SIGPLAN Workshop on Haskell*, pages 90–104, 2002.

[10] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, pages 315–326, 2007.

[11] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. *Journal of Functional Programming*, 12(6), 2002.

[12] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[13] J. Dolby and A. Chien. An evaluation of automatic object inline allocation techniques. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–20, 1998.

[14] I. Dragos and M. Odersky. Compiling generics through user-directed type specialization. In *Proc. Workshop on Implementation, Compilation, Optimization, of Object-Oriented Languages and Programming Systems*, pages 42–47, 2009.

[15] J. Feo and D. Cann. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10:349–366, 1990.

[16] C. Flanagan, A. Sabry, B. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 237–247, 1993.

[17] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005.

[18] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 130–141, 1995.

[19] J. Hughes. Why functional programming matters. *The Computer Journal*, 32: 98–107, 1989.

[20] G. Keller, M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, pages 261–272, 2010.

[21] A. Kennedy and D. Syme. Design and implementation of generics for the .NET common language runtime. In *Proc. ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1–12, 2001.

[22] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, 1994.

[23] D. Leijen and E. Meijer. Domain-specific embedded compilers. In *Proc. Conference on Domain-Specific Languages*, pages 109–122, 1999.

[24] X. Leroy. Unboxed objects and polymorphic typing. In *Proc. ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 177–188, 1992.

[25] X. Leroy. The effectiveness of type-based unboxing. Technical report, Boston College, 1997.

[26] G. Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, 1995.

[27] R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Transactions on Programming Languages and Systems*, 13(3):342–371, 1991.

[28] P. Hartel et al. Benchmarking implementations of functional languages with "pseudoknot", a float-intensive benchmark. *Journal of Functional Programming*, 6(4): 621–655, 1996.

[29] J. Peterson and M. Jones. Implementing type classes. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 227–236, 1993.

[30] S. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *Proc. European Symposium on Programming*, pages 18–44, 1996.

[31] S. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 636–666, 1991.

[32] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[33] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proc. 9th International Conference on Generative Programming and Component Engineering*, pages 127–136, 2010.

[34] S.-B. Scholz. Single Assignment C—efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 3(6):1005–1059, 2003.

[35] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, pages 51–62, 2008.

[36] Z. Shao. Flexible representation analysis. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, pages 85–98, 1997.

[37] J. Sipelstein and G. Blelloch. Collection-oriented languages. In *Proc. IEEE*, volume 9, pages 504–523, 1991.

[38] J. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. Liu, and W.-M. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical report, University of Illinois at Urbana-Champaign, 2012.

[39] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman, third edition, 2000.

[40] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[41] P. Thiemann. Unboxed values and polymorphic typing revisited. In *Proc. 7th International Conference on Functional Programming Languages and Computer Architecture*, pages 24–35, 1995.

[42] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Proc. ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[43] P. Wu, S. Midkiff, J. Moreira, and M. Gupta. Efficient support for complex numbers in Java. In *Proc. ACM Conference on Java Grande*, pages 109–118, 1999.