A Type Theory for Memory Allocation and Data Layout [*]

Leaf Petersen Robert Harper Karl Crary Frank Pfenning

Carnegie Mellon University

Abstract

Ordered type theory is an extension of linear type theory in which variables in the context may be neither dropped nor re-ordered. This restriction gives rise to a natural notion of *adjacency*. We show that a language based on ordered types can use this property to give an exact account of the layout of data in memory. The fuse constructor from ordered logic describes adjacency of values in memory, and the mobility modal describes pointers into the heap. We choose a particular allocation model based on a common implementation scheme for copying garbage collection and show how this permits us to separate out the allocation and initialization of memory locations in such a way as to account for optimizations such as the coalescing of multiple calls to the allocator.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors—*Compilers, Memory management*

General Terms

Languages, Theory

Keywords

Ordered Logic, Type Theory, Memory Management, Data Representation

[*]This material is based on work supported in part by NSF grants CCR-9984812 and CCR-0121633. Any opinions, findings, and conclusions or recommendations in this publication are those of the authors and do not reflect the views of this agency.

1 Introduction

High-level programming languages such as ML and Java allow programmers to program in terms of abstractions such as pairs, records, and objects, which have well-defined semantics but whose realizations in terms of the underlying concrete machine are left unspecified and unobservable.

Sometimes, it is necessary to program without these abstractions.

- A programmer may need to interact with an operating system or a network or another programming language in such a way as to require exact knowledge of, and control over, the manner in which data is laid out in memory.
- A compiler must choose a concrete implementation for the high-level abstractions provided by the source level language—such as the actual layout of data in memory and the manner in which such memory gets allocated and initialized.

Traditionally, both of these needs have been addressed in an un-typed, or a weakly typed fashion. Languages such as C give programmers relatively precise control over data layout and initialization at the expense of type and memory safety. Traditional compilers represent programs internally using un-typed languages, relying on the correctness of the compiler to preserve any safety properties enjoyed by the source program.

Recently, research in the areas of typed compilation and certified code [12, 21, 11] has focused on providing type systems for low-level languages in which abstractions such as control flow and data layout are made explicit. These ideas have been used in a number of compilers [12, 21, 9, 2, 19, 6]. However, some of the mechanisms that have been invented to describe low-level operations are fairly *ad hoc* and do not yet have an interpretation in standard type theory. For example, in the typed assembly language formalism[11], allocation and initialization can be separated, but at the expense of having to annotate each type with a flag indicating whether or not the value it classifies has been initialized. This kind of low-level technique seems unlikely to integrate well with a high-level programming language.

In this paper, we attempt to give a type theoretic account of data layout that provides a foundation for defining how high-level constructs such as pairs are laid out in memory. We realize our system with a concrete allocation model based on a common implementation of a copying garbage collector and show that we can separate out the process of allocating a block of memory from the process of initializing the individual memory words. Our system is flex-

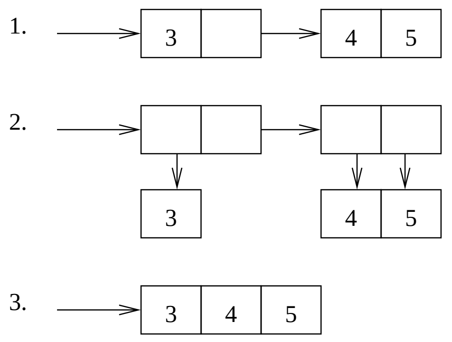

Figure 1. Three possible layouts for the term $(3,(4,5))$

ible enough to permit multiple allocation calls to be coalesced so that memory for multiple source level objects can be allocated simultaneously, while ensuring that calls to the allocator can never invalidate assumptions made about the state of partially initialized data.

An important contribution of this work is that it remains completely within the framework of a lambda calculus which enjoys the standard meta-theoretic properties. In this way, we reconcile the very low-level notion of allocated memory with the substitution properties expected of a high-level programming language. This is of particular interest because it suggests the possibility that these ideas could be made available to programmers, so that even programs requiring detailed control of memory layout could be written in a typed, high-level language.

2 Data layout and allocation

Specifying the layout of data in memory is an essential part of realizing a high-level program as a concrete collection of machine instructions and data, but one which is usually not of direct interest to programmers. The programmer cares about the ability to construct objects, but most of the time cares about the layout in memory only insofar as it affects the performance of operations on an object.

How terms should be laid out in memory is therefore a matter of policy for the compiler writer. For example, the lambda calculus term $(3,(4,5))$ of type $\texttt{int} \times (\texttt{int} \times \texttt{int})$ defines a pair whose first element is 3 and whose second element is a pair containing 4 and 5. Figure 1 shows several possible representations for this term. One compiler might choose to represent this as a pointer to a pair, whose elements are an integer and a pointer to another pair. However, another might choose to add an indirection to integers, or to attempt to flatten the whole term into three adjacent cells in memory.

The high level notion of pairing captures certain operational properties that are useful to the programmer, but does not uniquely specify an implementation strategy. Commonly, a compiler simply chooses to interpret the pair type as meaning one particular strategy. For the purposes of giving a general account of data layout, this is clearly unsatisfactory as it does not permit us to break the high-level concept into its constituent concepts.

A first step to a more general type theory for data layout is to observe that there seem to be two key concepts used by the different interpretations of pairing given in figure 1: adjacency and indirection. Each of the different choices of representation corresponds to a different choice as to which data is to be represented by *physically adjacent* bytes in memory and which data is to be represented via an *indirection* into another portion of memory. This is the first notion that we shall attempt to capture in our type system.

2.1 Allocation

Once the layout of data in memory has been made explicit, it becomes possible to consider the process by which new memory is created and initialized. We suggest that it is useful to think of this in terms of three stages, regardless of the mechanism employed.

Reservation is the process by which a new block of uninitialized memory is created.

Initialization is the process by which values get written into the reserved memory, potentially changing its type. It is important for type safety that either the memory be treated linearly in this stage, or else that the initialization operations be such that they only *refine* the type [3].

Allocation is the process by which a section of reserved (and presumably initialized) memory is made available as an ordinary unrestricted object.

Different memory-management systems combine these stages in different ways. For example, in the TAL framework [11], reservation and allocation are done atomically, and hence initialization is very restricted in how it can change the type.

The concrete memory management system that we choose to model is one commonly used in practice by copying garbage collectors and hence is of particular interest. This choice is not essential—other systems can be expressed using similar techniques to those we present here.

In a copying garbage collector, the available memory can be divided into two adjacent contiguous sections: a heap containing data that has been allocated since the last garbage collection (or perhaps just the youngest generation thereof), and a possibly empty freespace containing memory that has not yet been allocated. The allocator maintains an *allocation pointer* (or *freespace pointer*), which points to the end of the allocated data and the start of the free memory, and a *heap-limit pointer*, which points to the end of the free memory.

To create a new heap object requiring *n* bytes, the program first compares the allocation pointer to the heap-limit pointer to ensure that there are at least *n* bytes available in the freespace. If not, it calls the garbage collector to free up enough space. This step corresponds to the reservation phase discussed above. Once sufficient memory has been found—either in the existing freespace or by calling the garbage collector—the program may assume that *n* bytes of free space exist in front of the allocation pointer. We refer to this initialized area as the *frontier*.

Once space has been reserved on the frontier, values can be written into the individual cells of memory via offsets from the allocation pointer. This corresponds to the initialization phase.

At any point, the program may "move" a prefix of the frontier into the heap. The value of the allocation pointer becomes the pointer to the new heap value, and the allocation pointer is advanced past the allocated space. This corresponds to the allocation phase.

Figure 2 gives an example of this process. The first line shows a schematic diagram of the heap and the freespace, where a.p. stands for the allocation pointer and l.p. stands for the limit pointer. The

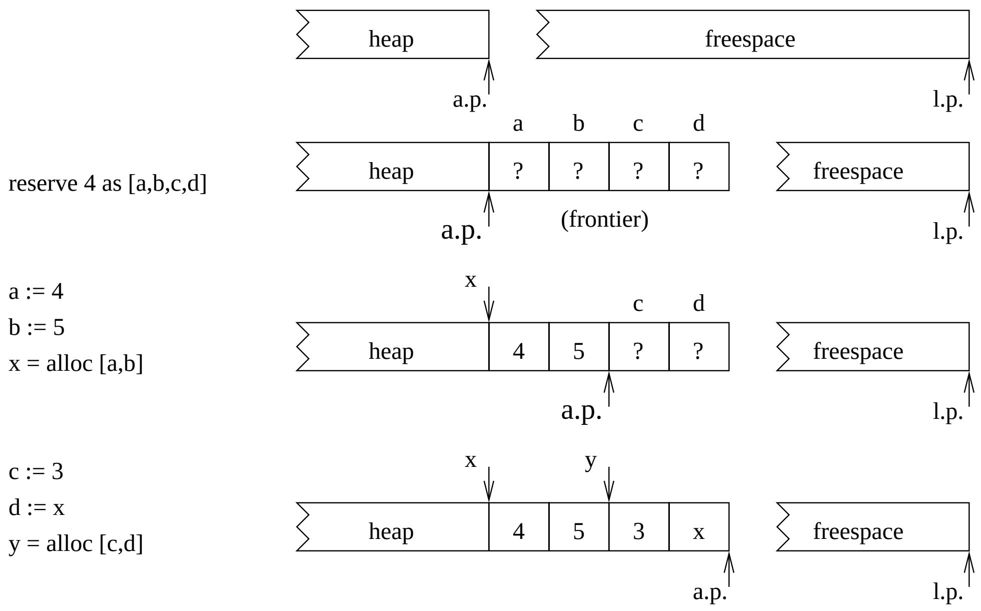

Figure 2. Reservation, initialization, and allocation of $(3,(4,5))$

ragged boundary of the freespace indicates that we have no information about its extent—it may potentially be exhausted.

The second line of the figure shows the result of reserving four words of space—sufficient for allocating the term $(3,(4,5))$ using the first layout strategy from Figure 1. We refer to the individual cells of the frontier by the names a,b,c and d. Note that this step may have invoked the garbage collector to free up more memory if the freespace from the previously line was in fact exhausted.

To create the pair $(4,5)$ we assign 4 to a, 5 to b, and then allocate a and b into the heap getting back a heap pointer x as shown on the third line of the figure. We can then initialize the top-level pair by writing 3 to c and x to d. A final allocation step gives us a pointer y which refers to a heap allocated structure of the form pictured in the first line of Figure 1.

As this example shows, we do not require that the entire frontier be allocated as a single object. The program may choose to reserve space for several objects at once and then initialize and allocate them individually. This optimization avoids multiple checks against the heap-limit pointer.

There are two constraints on this process that must be captured by our type system to ensure safety.

Firstly, the manner in which we "move" objects into the heap means that objects cannot be allocated from the middle or end of the frontier. Only prefixes of the frontier—that is, contiguous blocks of memory adjacent to the allocation pointer—may be allocated.

Secondly, reserved space in the frontier cannot persist across successive reservations nor across function calls. When the garbage collector is called it will copy the live data to a new heap and change the allocation pointer to point to this new location. Any partially initialized data that was previously in the frontier will be lost in the process.

This corresponds to a kind of destructive effect: the state of the frontier cannot be assumed to be preserved across the evaluation of any term that could potentially call the allocator. The type system must therefore ensure that no assumptions about the state of the frontier can persist across the evaluation of any term that might reserve or allocate memory.

3 Ordered linear type theory

Ordered (or non-commutative) linear logic is a variant of standard linear logic in which hypotheses must not only be used exactly once, but must also be used in order [17, 16, 18, 15]. The corresponding proof terms make up an ordered lambda calculus that is characterized by the lack of an exchange property for the ordered context in addition to the usual linearity restrictions. We present a small fragment of the ordered lambda calculus by way of introduction to the these ideas. The presentation here is simpler than previous work, in that it omits the linear context, retaining only the ordered and unrestricted contexts. The modal therefore moves directly from the ordered terms to unrestricted terms.

Typing rules for the ordered lambda calculus have the form $\Gamma;\Omega \vdash M:\tau$, indicating that the M has type τ under the variable assumptions declared in the unrestricted context Γ and the ordered context Ω. We distinguish syntactically between ordered variables a which must be used linearly and in order, and unrestricted variables x which may be used arbitrarily often.

$$\frac{}{\Gamma;a{:}\tau \vdash a:\tau} \qquad \frac{}{\Gamma,x{:}\tau,\Gamma';\cdot \vdash x:\tau}$$

Unlike standard linear type theory, the ordered comma operator Ω_1,Ω_2 is interpreted as simple list concatenation and does not permit the intermingling of hypotheses. Where unambiguous, we write $a{:}\tau$ instead of $a{:}\tau,\cdot$ for singleton contexts.

$$(\Omega_1,\Omega_2) \overset{\text{def}}{=} \begin{cases} \Omega_2 & \text{if } \Omega_1 = \cdot \\ a{:}\tau,(\Omega_1',\Omega_2) & \text{if } \Omega_1 = a{:}\tau,\Omega_1' \end{cases}$$

$$
\begin{array}{lcll}
\tau & ::= & \texttt{int} & \text{integers} \\
 & | & \tau_1 \to \tau_2 & \text{unrestricted arrow} \\
 & | & \tau_1 \bullet \tau_2 & \text{ordered multiplicative} \\
 & | & !\,\tau & \text{modal type} \\
\\
\Omega & ::= & \cdot \mid a{:}\tau, \Omega & \text{ordered contexts} \\
\Gamma & ::= & \cdot \mid x{:}\tau, \Gamma & \text{unrestricted contexts} \\
\\
M & ::= & a & \text{ordered variables} \\
 & | & x & \text{unrestricted variables} \\
 & | & \overline{n} & \text{integer literals} \\
 & | & M \bullet M & \text{fuse intro} \\
 & | & \texttt{let}\, a_1 \bullet a_2 = M \,\texttt{in}\, M & \text{fuse elim} \\
 & | & \lambda(x{:}\tau).E & \text{lambda intro} \\
 & | & M\ M & \text{lambda elim} \\
 & | & !\,M & \text{modal intro} \\
 & | & \texttt{let}\,!\,x = M \,\texttt{in}\, M & \text{modal elim}
\end{array}
$$

Figure 3. Standard ordered lambda calculus syntax

This definition means that concatenation of contexts preserves the order of the entries in the contexts.

The multiplicative connective (fuse) demonstrates a use of this concatenation operator.

$$\frac{\Gamma;\Omega_1 \vdash M_1 : \tau_1 \quad \Gamma;\Omega_2 \vdash M_2 : \tau_2}{\Gamma;\Omega_1,\Omega_2 \vdash M_1 \bullet M_2 : \tau_1 \bullet \tau_2}$$

The elimination rule for fuse splits it into components and places them in the ordered context. Notice that the variables representing the components of M_1 go into the ordered context in place of Ω.

$$\frac{\Gamma;\Omega \vdash M_1 : \tau_1 \bullet \tau_2 \quad \Gamma;\Omega_L, a_1{:}\tau_1, a_2{:}\tau_2, \Omega_R \vdash M_2 : \tau}{\Gamma;\Omega_L,\Omega,\Omega_R \vdash \texttt{let}\, a_1 \bullet a_2 = M_1 \,\texttt{in}\, M_2 : \tau}$$

Finally, the mobility modal permits terms that are orderedly closed to be moved to the unrestricted context.

$$\frac{\Gamma;\cdot \vdash M : \tau}{\Gamma;\cdot \vdash !M : !\tau} \qquad \frac{\Gamma;\Omega \vdash M : !\tau \quad \Gamma, x{:}\tau;\Omega_L,\Omega_R \vdash M' : \tau'}{\Gamma;\Omega_L,\Omega,\Omega_R \vdash \texttt{let}\,!x = M \,\texttt{in}\, M' : \tau'}$$

3.1 Size preservation and adjacency

There are three interesting observations that we can make about ordered lambda calculus terms that motivate the application of ordered type theory to data layout.

1. Because ordered variables may not exchange position in the context, we may think of ordered variables as simply standing for *locations* in the ordered context.
2. We may break ordered terms down into their components and re-form them, but we may not change their order. In particular, the term that splits apart an ordered pair and reforms it in the opposite order is not well-typed.

   $$\texttt{let}\ a_1 \bullet a_2 = a\ \texttt{in}\ a_2 \bullet a_1$$

   Viewed as a linear (rather than ordered) term, this code would be well-typed.
3. The ! modality takes an ordered term whose location is fixed and moves it into the unrestricted context, where its location become indeterminate.

Based on these observations, we propose the following three intuitions as the basis for our system.

1. An ordered context may be thought of as describing a particular region of memory under consideration. Ordered variables correspond to locations, or offsets into the region. Adjacent variables in the context correspond to physically adjacent locations, with extents given by the types of the variables.
2. The fuse constructor $\tau_1 \bullet \tau_2$ describes terms that are physically adjacent in memory. The fact that we cannot reorder ordered terms corresponds naturally to the fact that we cannot reorder bytes in memory.
3. The ! modality $!\,\tau$ corresponds to an indirection out of the region of memory described by the ordered context into another (unspecified) part of the heap.

The standard ordered lambda calculus does not entirely justify these intuitions. Ordered terms preserve the *order* of sub-components, but they do not in general preserve their *adjacency*. The essence of this problem can be seen in the derived ordered substitution principle.

$$\frac{\Gamma;\Omega \vdash M : \tau \quad \Gamma;\Omega_1, a{:}\tau, \Omega_2 \vdash M' : \tau'}{\Gamma;\Omega_1,\Omega,\Omega_2 \vdash \texttt{let}\, a = M \,\texttt{in}\, M' : \tau'}$$

Notice that the portion of the ordered context that is passed to the term being bound is replaced with the variable itself when type-checking the rest of the body. Our intention is that operations such as this should be done in-place on the memory described by the ordered context. However, the following term demonstrates that this does not hold in the general ordered lambda calculus.

$$\frac{\Gamma;\cdot \vdash 3 : \texttt{int} \quad \Gamma;\Omega_1, a{:}\texttt{int}, \Omega_2 \vdash M' : \tau'}{\Gamma;\Omega_1,\cdot,\Omega_2 \vdash \texttt{let}\, a = 3 \,\texttt{in}\, M' : \tau'}$$

The problem is that we are able to insert unrestricted terms into the ordered terms in arbitrary places. While this does not violate our notion that ordered variables correspond to locations, it does mean that these locations are not fixed. Operationally, it would seem that we would be forced to shift all of Ω_2 over in memory to make room for the new term in the context.

An alternative way of looking at this is that the general ordered lambda calculus is not *size preserving*: the sub-derivation $\Gamma;\cdot \vdash 3 : \texttt{int}$ produces a term of size one from a context of size zero. If we interpret the ordered context as describing a region of memory, then the above term inserts a word-sized value into an empty region of memory! In order to prevent such problematic terms, it is necessary to carefully restrict the calculus in such a way as to ensure that operations on memory preserve size.

The notion of size preservation is the last insight necessary to formulate a lambda calculus in which we can give a full account for data layout. We will use the fuse type to describe adjacency and the modal type to describe indirection, while restricting the terms in such a way as to enforce various key size preservation properties. The allocation model described in section 2 will be accounted for by using an ordered context to describe the frontier. Ordered variables then become offsets into the frontier, and reservation, initialization, and allocation become operations on ordered terms. The linearity of

$$
\begin{array}{lcl}
k & ::= & \mathtt{T}_{\text{reg}} \mid \mathtt{T}_{\text{h}} \\
\tau & ::= & 1 \mid \mathtt{int} \mid \tau_1 \to \tau_2 \mid \tau_1 \bullet \tau_2 \mid\ !\tau \mid \mathtt{NS} \\
\\
Q & ::= & a \mid * \mid Q \bullet Q \\
V & ::= & * \mid \mathtt{ns} \mid \overline{n} \mid V \bullet V \mid \lambda(x{:}\tau).E \mid\ !V \\
M & ::= & x \mid \mathtt{ns} \mid \overline{n} \mid \lambda(x{:}\tau).E \mid\ !V \\
E & ::= & \mathtt{ret}\,M \mid M\,M \mid \mathtt{let}\,x : \tau = E\ \mathtt{in}\ E \\
 & & \mid \mathtt{reserve}_n\ \mathtt{as}\ a\ \mathtt{in}\ E \mid \mathtt{alloc}\ Q\ \mathtt{as}\ x\ \mathtt{in}\ E \\
 & & \mid Q := M\ \mathtt{as}\ a\ \mathtt{in}\ E \\
 & & \mid \mathtt{let}\ a = Q\ \mathtt{in}\ E \mid \mathtt{let}\ a \bullet a = Q\ \mathtt{in}\ E \\
 & & \mid \mathtt{let}\ * = Q\ \mathtt{in}\ E \\
 & & \mid \mathtt{let!}\ (x \bullet x) = M\ \mathtt{in}\ E \mid \mathtt{let}\ !x = M\ \mathtt{in}\ E \\
\\
\Gamma & ::= & \cdot \mid x{:}\tau, \Gamma \\
\Omega & ::= & \cdot \mid a{:}\tau, \Omega \\
\omega & ::= & \cdot \mid a \mapsto V, \omega
\end{array}
$$

Figure 4. Syntax

the ordered context will permit destructive operations on the frontier (such as initialization), and the size preservation property will ensure that all operations on the frontier may be done in-place.

4 The orderly lambda calculus

We now have all of the ideas that we need to define a language for data layout and allocation, which we shall call the *orderly lambda calculus*, or λ^{ord} for short. For the sake of brevity, this paper will focus on a small core language that captures the essential ideas.

The syntax of the core language is given in figure 4. We use the notation τ^n for an n-ary fuse of τ.

$$
\begin{array}{lcl}
\tau^0 & = & 1 \\
\tau^{n+1} & = & \tau \bullet \tau^n
\end{array}
$$

For data layout purposes, we only require a few new types from the ordered lambda calculus: the fuse constructor which models adjacency; the modal constructor, which models indirection; and the multiplicative unit. Other types include a base type of integers and the type of unrestricted functions. The NS (nonsense) type is the type of a single un-initialized word of memory.

It is important for our purposes to distinguish between types which are of unit size and hence can be kept in registers or on the stack, and other types that must be heap allocated. This is accomplished by a kinding distinction $\vdash \tau : k$. The kind $\mathtt{T}_{\text{reg}}$ classifies the types of values which may be loaded into registers, whereas the kind $\mathtt{T}_{\text{h}}$ classifies types that may be heap-allocated (a strict super-set of the former).

An important property of this language is that types uniquely determine the size of the data they classify.

$$
|\tau| \overset{\text{def}}{=} \begin{cases} 0 & \text{if } \tau = 1 \\ |\tau_1| + |\tau_2| & \text{if } \tau = \tau_1 \bullet \tau_2 \\ 1 & \text{if } \tau = \tau_1 \to \tau_2,\ \mathtt{int},\ \mathtt{NS} \text{ or } !\tau' \end{cases}
$$

For simplicity, the smallest unit of size we consider is a single machine word. The multiplicative unit type has size zero, since it is inhabited by a single value which therefore does not need to be represented. We view the function type as having unit size, since we expect that a practical implementation would use closures to represent functions. Under closure conversion, lambdas become existentially quantified records allocated on the heap, and hence are represented by a pointer of unit size. We assume that the actual code for the function will be statically allocated.

Ordered contexts Ω map ordered variables a to types τ, and are used to describe regions of memory (in particular, the frontier). The notion of sizing for types extends naturally to ordered contexts.

$$
|\Omega| \overset{\text{def}}{=} \begin{cases} 0 & \text{if } \Omega = \cdot \\ |\tau| + |\Omega'| & \text{if } \Omega = a{:}\tau, \Omega' \end{cases}
$$

As before, exchanging, discarding, or duplicating variables in the ordered context is not permitted.

Unrestricted contexts Γ map ordinary variables x to their types. The well-formedness judgement for unrestricted contexts checks that all unrestricted variables have unit-sized types—that is, types whose kind is $\mathtt{T}_{\text{reg}}$. Ordinary variables correspond to registers or stack slots in the underlying machine, and so are restricted to have word size via this kinding mechanism. This is a key point about the orderly lambda calculus: all large objects are required to be explicitly allocated and initialized.

The term level of λ^{ord} is split into four separate syntactic classes: *coercion terms* Q, *heap values* V, *terms* M and *expressions* E. The main typing judgements are described in figure 5, along with comments about the size properties which they enjoy. Complete definitions of the typing rules can be found in appendix A.

Making allocation explicit introduces a kind of effect into the language. Reserving and allocating memory is an effectful operation, and as we saw in the previous section these effects may interfere. In order to control these effects and their interaction we introduce a distinction between terms M and expressions E in the style of Pfenning and Davies [14], but without an explicit modal type for computations. (The computation type does not seem useful in our setting since we do not have the inclusion of expressions into terms, instead taking the partial arrow as primitive).

The syntactic form we impose is not overly restrictive: it is actually related to, but more permissive than, the A-normal or CPS forms that many compilers typically use.

4.1 Terms

Terms M correspond to values that do not reserve or allocate in the course of their evaluation, but that may contain free references to ordered variables (that is, to the frontier). In this presentation, all terms are values—but it is straightforward and useful to include other primitive operations that do not allocate (such as integer operations) at this level. The typing judgement for terms is of the form $\Gamma;\Omega \vdash_{\text{trm}} M : \tau$. The term M may refer to variables in Γ arbitrarily often, but *must* refer to each variable in Ω exactly once, and in an ordered fashion.

The typing rules for terms are for the most part unsurprising. For the λ-abstraction case, the body of the function is checked as an expression, with the argument placed in the unrestricted context.

$$
\frac{\Gamma, x{:}\tau; \Omega \vdash_{\text{exp}} E : \tau'}{\Gamma; \Omega \vdash_{\text{trm}} \lambda(x{:}\tau).E : \tau \to \tau'}
$$

Notice that we permit free references to the frontier in functions. Since function application lies in the category of expressions, we

Judgement	*Size properties*	*Meaning*
$\vdash \Omega$		Ω is a well-formed ordered context.
$\vdash \Gamma$	$\forall x \in \Gamma, \lvert\Gamma(x)\rvert = 1$	Γ is a well-formed unrestricted context.
$\vdash \tau : k$	if $k = \mathtt{T}_{\text{reg}}$ then $\lvert\tau\rvert = 1$	τ is a well-formed type.
$\Omega \vdash_{\text{crc}} Q : \tau$	$\lvert\Omega\rvert = \lvert\tau\rvert$	Q coerces Ω to look like τ .
$\Gamma;\Omega \vdash_{\text{trm}} M : \tau$	$\lvert\tau\rvert = 1$	M is a non-allocating/non-reserving term of type τ.
$\Gamma;\Omega \vdash_{\text{exp}} E : \tau$	$\lvert\tau\rvert = 1$	E is a well typed expression of type τ which consumes Ω.
$\vdash_{\text{val}} V : \tau$	$\lvert V\rvert = \lvert\tau\rvert$	V is a closed value of type τ.
$\vdash \omega : \Omega$	$\lvert\Omega\rvert = \lvert\omega\rvert$	ω is a well-typed frontier for the ordered context Ω.

Figure 5. Typing judgements for λ^{ord}

will defer discussion of the elimination form to Section 4.4. All other terms must be closed with respect to the ordered context.

The most non-standard term is $!V$. This term corresponds to a pointer into the heap to a location occupied by the heap value V, and is the canonical form for terms of type $!\tau$.

$$\frac{\vdash_{\text{val}} V : \tau}{\Gamma; \cdot \vdash_{\text{trm}} !V : !\tau}$$

An interesting facet of our presentation is that we account for heap allocation without requiring an explicit heap (for example in the style of Morrisett and Harper [10]). In a heap semantics, a pointer to a value V is represented by a label ℓ, with ℓ bound to V in an explicit heap data-structure. Since sharing is not observable in our simple calculus, we avoid this extra complexity by representing such values directly as $!V$, denoting a pointer to a location occupied by V. We stress that this is purely a technical convenience—it is straightforward to give a heap semantics in which the sharing is made explicit in the usual fashion.

4.2 Heap Values

Heap values V represent terms that may occur in memory. It is therefore essential that they be closed. An open heap term would require that a new copy be implicitly allocated every time different values were substituted into it, which is contrary to the aims of λ^{ord}. The typing judgement for heap values, $\vdash_{\text{val}} V : \tau$, enforces this property.

The primary motivation for having heap values comes from the operational semantics of the language. However, it is not intended that they should play the role of so-called "semantic objects" that are only permitted to be introduced in the course of evaluation. It is perfectly reasonable for a programmer to write heap values in the source program. Doing so corresponds precisely to the notion of statically allocated data—that is, data that is present in the heap at the start of the program.

The important difference between heap values and terms is that heap values may be of arbitrary size. This is reflected in the syntax by the value $V_1 \bullet V_2$, denoting a contiguous block of memory in which V_1 is laid out adjacent to the value V_2.

The fact that fused terms are adjacent means that the $\bullet$ constructor is associative in the sense that the term $3 \bullet (4 \bullet 5)$ has the same representation in memory as the term $(3 \bullet 4) \bullet 5$. Both terms describe three successive words of memory, occupied by the integers 3, 4, and 5 respectively. This is a fundamental difference from ordinary lambda calculus pairing, in which $(3,(4,5))$ is almost certain to have a different representation from $((3,4),5)$.

This associativity is just one example of values which have different types but the same representations. Other examples include values involving the ordered unit, $*$. Since we do not choose to represent this value, we expect that the representations of $3 \bullet *$, $* \bullet 3$, and 3 will all be the same at runtime.

Coercion terms exist to provide a mechanism by which to convert between such values which have different typing structure but the same underlying representation.

4.3 Coercions

The level of coercion terms in this fragment of the language is extremely simple, consisting only of variables a, the ordered unit $*$, and fuse $Q_1 \bullet Q_2$. Coercion binding and elimination forms are provided at the expression level (Section 4.4).

Intuitively, coercion terms package up the frontier into new forms without changing the underlying representation. For example, the term $a_1 \bullet a_2$ takes the section of the frontier described by a_1 and the section described by a_2 and combines them into a single fuse which could then be bound at a new name using the expression level coercion `let`. The orderedness of the terms ensures that the two sections were already adjacent, and hence combining them into a fuse does not change their representation.

The typing judgement for coercion terms is of the form $\Omega \vdash_{\text{crc}} Q : \tau$, signifying that Q re-associates Ω to have the form τ. The coercive nature of the terms is exhibited in the size preservation property that holds of this judgement: that $\lvert\Omega\rvert = \lvert\tau\rvert$.

$$\frac{}{a{:}\tau \vdash_{\text{crc}} a : \tau} \qquad \frac{\Omega_1 \vdash_{\text{crc}} Q_1 : \tau_1 \quad \Omega_2 \vdash_{\text{crc}} Q_2 : \tau_2}{\Omega_1, \Omega_2 \vdash_{\text{crc}} Q_1 \bullet Q_2 : \tau_1 \bullet \tau_2}$$

The unit term is well-typed in the empty context.

$$\frac{}{\cdot \vdash_{\text{crc}} * : 1}$$

4.4 Expressions

So far we have only seen the value forms that occupy or coerce memory, but that do not modify it. The memory operations—reservation, allocation, and initialization—are all done at the level of expressions.

The well-formedness judgement for expressions is given by $\Gamma;\Omega \vdash_{\text{exp}} E : \tau$. The ordered context Ω in the typing judgement describes the current state of the frontier. Because of the destructive nature of the reserve and allocate operations, the interpretation is that the frontier is *consumed* by the expression E. That is, any space that is on the frontier must either be allocated by E, or explicitly destroyed.

As we saw in section 2, memory operations are effectful, and so the type system for expressions must be carefully designed to ensure that these effects do not interfere. This is enforced by always passing the entire ordered context (and hence the entire frontier) to each sub-*expression* (but not sub-*term*). In this way, we ensure that every possibly allocating/reserving expression has a correct view of the entire frontier when it is evaluated.

The expressions can be conceptually divided into four basic categories.

Ordinary expressions

The inclusion of values into expressions is given by the expression $\mathtt{ret}\, M$.

$$\frac{\Gamma; \cdot \vdash_{\text{trm}} M : \tau}{\Gamma; \cdot \vdash_{\text{exp}} \mathtt{ret}\, M : \tau}$$

This is the only value form for expressions, and consumes no resources. It is unsound to permit the term M to contain ordered variables, since it may be substituted for an unrestricted variable by the primitive let form discussed below.

Function application is an expression, since the evaluation of the body of the function may engender memory effects. Applications are syntactically restricted to permit only application of a term to another term.

$$\frac{\Gamma; \Omega \vdash_{\text{trm}} M_1 : \tau_1 \to \tau_2 \quad \Gamma; \cdot \vdash_{\text{trm}} M_2 : \tau_1}{\Gamma; \Omega \vdash_{\text{exp}} M_1\, M_2 : \tau_2}$$

The term being applied is permitted to refer to ordered variables, but the argument must be closed since unrestricted functions may duplicate or drop their arguments. Application allows us to define a term-level `let` construct with the following derived typing rule.

$$\frac{\Gamma; \cdot \vdash_{\text{trm}} M : \tau \quad \Gamma, x{:}\tau; \Omega \vdash_{\text{exp}} E : \tau'}{\Gamma; \Omega \vdash_{\text{exp}} \mathtt{let}\, x{:}\tau = M\, \mathtt{in}\, E : \tau'}$$

This `let` is not fully general, since there is no way to bind the result of an application to a variable. Therefore, we introduce a primitive let form to bind expressions to variables.

$$\frac{\Gamma; \Omega \vdash_{\text{exp}} E_1 : \tau_1 \quad \Gamma, x{:}\tau_1; \cdot \vdash_{\text{exp}} E_2 : \tau_2}{\Gamma; \Omega \vdash_{\text{exp}} \mathtt{let}\, x{:}\tau_1 = E_1\, \mathtt{in}\, E_2 : \tau_2}$$

Notice that we pass the entire ordered context to the first subexpression. This is a crucial point: E_1 may have memory effects that could invalidate any previous assumptions about the state of the frontier that E_2 might make. Therefore, E_2 cannot assume anything at all about the state of the frontier—that is, it must be well-typed in an empty ordered context.

Somewhat surprisingly, it is safe to permit E_1 to have free references to the ordered context. This is reasonable because expressions *consume* resources, but do not *contain* them. By this we mean that the value form for expressions ($\mathtt{ret}\, M$) is well-typed only in an empty ordered context. Therefore, if the ordered context Ω is not empty, then E_1 must explicitly destroy or allocate all of the memory described by Ω before it reaches a value. Since this value will be orderedly closed, it is safe to substitute it freely for the unrestricted variable x.

Memory expressions

The most interesting and non-standard expressions are those dealing directly with the frontier. Recall that there are three operations of interest: reserving space on the frontier, initializing pieces of the frontier, and allocating prefixes of the frontier into the heap. These three operations are captured directly as primitives. As we shall see later, this is not entirely necessary—by extending the type system somewhat we can give types to these primitives as constants. For simplicity however, we first present them as primitive notions.

The first operation, reservation, discards any resources that were previously mentioned in the ordered context, and introduces n words of nonsense into the frontier.

$$\frac{\Gamma; a{:}\mathtt{NS}^n \vdash_{\text{exp}} E : \tau}{\Gamma; \Omega \vdash_{\text{exp}} \mathtt{reserve}_n\, \mathtt{as}\, a\, \mathtt{in}\, E : \tau}$$

This corresponds exactly to the reservation operation described in Section 2.1, which destroys any existing data on the frontier and provides a block of "new" uninitialized space.

Memory must be written using assignment.

$$\frac{\begin{array}{c}\Omega \vdash_{\text{crc}} Q : \tau \quad \vdash \tau : \mathtt{T}_{\text{reg}} \\ \Gamma; \cdot \vdash_{\text{trm}} M : \tau' \quad \Gamma; \Omega_L, a{:}\tau', \Omega_R \vdash_{\text{exp}} E : \tau''\end{array}}{\Gamma; \Omega_L, \Omega, \Omega_R \vdash_{\text{exp}} Q := M\, \mathtt{as}\, a\, \mathtt{in}\, E : \tau''}$$

The ordered term Q gives the location in the ordered context to which the value should be written. This location is then referred to by a in the body of the expression. The linearity of the ordered context is important here, since we are destructively changing the type of a memory location.

At any point, space can be allocated from the left side of the frontier with the `alloc` construct.

$$\frac{\Omega_L \vdash_{\text{crc}} Q : \tau \quad \Gamma, x{:}!\tau; \Omega_R \vdash_{\text{exp}} E : \tau'}{\Gamma; \Omega_L, \Omega_R \vdash_{\text{exp}} \mathtt{alloc}\, Q\, \mathtt{as}\, x\, \mathtt{in}\, E : \tau'}$$

The coercion term Q describes a section of the frontier to be packaged up as a boxed heap value. The splitting of the ordered context ensures that the term to be allocated is a prefix of the frontier. The new heap value is given a pointer type and permitted to be used unrestrictedly for the rest of the program.

Coercion expressions

The memory expressions manipulate the frontier using ordered variables, which stand for offsets into the frontier. Coercions are used to manipulate ordered variables, combining them into bigger terms or breaking them into smaller pieces.

The simplest coercion expression is the elimination form for unit.

$$\frac{\Omega \vdash_{\text{crc}} Q : 1 \quad \Gamma; \Omega_L, \Omega_R \vdash_{\text{exp}} E : \tau}{\Gamma; \Omega_L, \Omega, \Omega_R \vdash_{\text{exp}} \mathtt{let} * = Q\, \mathtt{in}\, E : \tau}$$

$$\frac{\Gamma;\cdot\vdash_{\text{trm}} M:!\tau \quad \vdash \tau[i]:\mathrm{T}_{\text{reg}} \quad \Gamma,x{:}\tau[i];\Omega\vdash_{\text{exp}} E:\tau_2}{\Gamma;\Omega\vdash_{\text{exp}} \mathtt{load}_\tau\ x=M[i]\ \mathtt{in}\ E:\tau_2} \qquad \text{where} \quad \tau[i]\overset{\text{def}}{=}\begin{cases}\tau_1[i] & \text{if } \tau=\tau_1\bullet\tau_2 \text{ and } |\tau_1|>i\\ \tau_2[i-|\tau_1|] & \text{if } \tau=\tau_1\bullet\tau_2 \text{ and } |\tau_1|\le i\\ \tau & \text{if } \tau \text{ is not a fuse and } i=0\end{cases}$$

$$\mathtt{load}_\tau\ x=M[i]\ \mathtt{in}\ E\overset{\text{def}}{=}\begin{cases}\mathtt{let!}\ x_1\bullet x_2=M\ \mathtt{in} & \\ \qquad \mathtt{load}_{\tau_1}\ x=x_1[i]\ \mathtt{in}\ E & \text{if } \tau=\tau_1\bullet\tau_2 \text{ and } |\tau_1|>i\\ \mathtt{let!}\ x_1\bullet x_2=M\ \mathtt{in} & \\ \qquad \mathtt{load}_{\tau_2}\ x=x_2[i-|\tau_1|]\ \mathtt{in}\ E & \text{if } \tau=\tau_1\bullet\tau_2 \text{ and } |\tau_1|\le i\\ \mathtt{let}\ !x=M\ \mathtt{in}\ E & \text{if } |\tau|=1 \text{ and } \tau \text{ is not a fuse}\end{cases}$$

Figure 6. An example of a direct-load defined in terms of split

Since the unit term is considered to have zero size, we may eliminate it freely from the ordered context without changing the size or adjacency properties of the terms in the frontier.

The elimination form for fuse is also a coercion expression.

$$\frac{\Omega\vdash_{\text{crc}} Q:\tau_1\bullet\tau_2 \quad \Gamma;\Omega_1,a_1{:}\tau_1,a_2{:}\tau_2,\Omega_2\vdash_{\text{exp}} E:\tau}{\Gamma;\Omega_1,\Omega,\Omega_2\vdash_{\text{exp}} \mathtt{let}\,a_1\bullet a_2=Q\,\mathtt{in}\,E:\tau}$$

The intuition is that since $\tau_1\bullet\tau_2$ describes two adjacent blocks of memory, we are free to view the single block of memory described by Q as two adjacent blocks at offsets named by a_1 and a_2.

The last coercion operation is the simple ordered let form, which permits ordered terms to be packaged up or renamed.

$$\frac{\Omega\vdash_{\text{crc}} Q:\tau \quad \Gamma;\Omega_1,a{:}\tau,\Omega_2\vdash_{\text{exp}} E:\tau'}{\Gamma;\Omega_1,\Omega,\Omega_2\vdash_{\text{exp}} \mathtt{let}\,a=Q\,\mathtt{in}\,E:\tau'}$$

Load expressions

The memory operations account for the creation of heap objects. Equally important is the ability to load values out of the heap. Once an object is in the heap, we must have some way of accessing its components. Pointers to "small" objects can be de-referenced directly.

$$\frac{\Gamma;\cdot\vdash_{\text{trm}} M:!\tau_1 \quad \vdash\tau_1:\mathrm{T}_{\text{reg}} \quad \Gamma,x{:}\tau_1;\Omega\vdash_{\text{exp}} E:\tau_2}{\Gamma;\Omega\vdash_{\text{exp}} \mathtt{let!}\,x=M\,\mathtt{in}\,E:\tau_2}$$

The kinding restriction ensures that the only values that can be loaded with this operation are those that will fit into a register.

To access the fields of larger objects, we provide a composite elimination construct that takes a pointer to a large object, and produces two pointers to the immediate subcomponents of the object.

$$\frac{\begin{array}{c}\Gamma;\cdot\vdash_{\text{trm}} M:!(\tau_1\bullet\tau_2)\\ \Gamma,x_1{:}!\tau_1,x_2{:}!\tau_2;\Omega\vdash_{\text{exp}} E:\tau\end{array}}{\Gamma;\Omega\vdash_{\text{exp}} \mathtt{let!}(x_1\bullet x_2)=M\,\mathtt{in}\,E:\tau}$$

Notice that the variables are bound not to the components of M themselves, but rather to *pointers* to the components of M. Using this expression we may successively iterate over large composite objects until we arrive at a pointer to a small object which can be loaded directly.

This construct is somewhat disturbing from a practical standpoint for two reasons. In the first place, it seems to require pointers into the interior of objects (sometimes called locatives) in order to be implemented efficiently. While not completely out of the question, interior pointers can be quite problematic for copying garbage collectors (at least when implemented as direct pointers into the interior of heap objects).

More importantly however, this construct does not permit constant time access to fields of a heap-allocated record. For example, to access the last element of a n-ary tuple in right-associated form requires n computations before we arrive at a term that can be loaded directly. This is clearly impractical.

We choose to use this "split" operation as the primitive notion because it provides a simple and natural elimination form. In practice however, it is likely that this term would be eliminated in favor of one of a number of direct-load constructs that are definable in terms of split (figure 6). By taking such a direct-load as primitive and giving it a direct implementation, the need for the interior pointers is eliminated and fields of records can be loaded in constant time.

4.5 Frontier semantics

In order to make the connection between the orderly lambda calculus and the frontier model of allocation clear, the semantics keeps an explicit frontier. This means that the reduction relation is defined not just on expressions, but rather on a frontier and an expression together.

Frontier terms ω (as defined in figure 4) map ordered variables (that is, offsets) to values V. From the standpoint of the operational semantics, the frontier plays a role very similar to an explicit substitution. The typing judgement for the frontier, $\vdash\omega:\Omega$, asserts that the ordered context Ω describes a frontier that looks like ω.

$$\frac{}{\vdash\cdot:\cdot} \qquad \frac{\vdash_{\text{val}} V:\tau \quad \vdash\omega:\Omega \quad (a\notin\Omega)}{\vdash(a\mapsto V,\omega):(a{:}\tau,\Omega)}$$

The evaluation relation for the orderly lambda calculus is given in terms of frontier/expression pairs.

$$\frac{\vdash\omega:\Omega \quad \cdot;\Omega\vdash_{\text{exp}} E:\tau}{\vdash(\omega,E):\tau}$$

The relation $(\omega,E)\mapsto(\omega',E')$ indicates that in frontier ω, the expression E reduces in a single step to the expression E', with new frontier ω'. The complete definition of this relation is given in Appendix B.

It is straightforward to show that reduction preserves typing, and that well-typed terms that are not values may always be reduced further.

Theorem 1 (Progress & Preservation)
If $\vdash (\omega, E) : \tau$ *then*

1. *Either* $(\omega, E) \mapsto (\omega', E')$ *or* E *is a value.*
2. *if* $(\omega, E) \mapsto (\omega', E')$ *then* $\vdash (\omega', E') : \tau$

PROOF. The proof proceeds by induction on the derivation of $\cdot; \Omega \vdash_{\text{exp}} E : \tau$, with the help of several substitution lemmas and some auxiliary lemmas proving properties of ordered contexts and frontiers. □

4.6 Size properties

An important property of the orderly lambda calculus is that types uniquely determine the size of the data that they represent. We have informally mentioned a number of sizing properties of the calculus: in particular that coercion terms preserve size, and that terms and expressions are always of unit size (so that they can be kept in registers).

These properties can be formalized as follows.

Theorem 2 (Size)

1. *If* $\vdash \tau : \mathtt{T}_{reg}$ *then* $|\tau| = 1$
2. *If* $\vdash \tau : \mathtt{T}_{h}$ *then* $\exists i$ *such that* $|\tau| = i$
3. *If* $\Omega \vdash_{crc} Q : \tau$ *then* $|\Omega| = |\tau|$
4. *If* $\vdash_{val} V : \tau$ *then* $|V| = |\tau|$
5. *If* $\Gamma; \Omega \vdash_{trm} M : \tau$ *then* $|\tau| = 1$
6. *If* $\Gamma; \Omega \vdash_{exp} E : \tau$ *then* $|\tau| = 1$
7. *If* $\vdash \omega : \Omega$ *then* $|\Omega| = |\omega|$.

PROOF. For each clause we proceed separately by induction on typing derivations. □

5 Representing the lambda calculus

One of the intended uses of λ^{ord} is as a target language for translation from higher-level languages. To show how this can be done, and to provide some intuition into how the language is used, we present in this section a translation from the simply typed lambda calculus with products and unit into the orderly lambda calculus.

We begin by defining a translation $\ulcorner \tau \urcorner$ that maps each ordinary lambda calculus type to a λ^{ord} type.

$$\begin{aligned}
\ulcorner \mathtt{int} \urcorner &= \mathtt{int} \\
\ulcorner \mathtt{unit} \urcorner &= !1 \\
\ulcorner \tau_1 \to \tau_2 \urcorner &= \ulcorner \tau_1 \urcorner \to \ulcorner \tau_2 \urcorner \\
\ulcorner \tau_1 \times \tau_2 \urcorner &= !(\ulcorner \tau_1 \urcorner \bullet \ulcorner \tau_2 \urcorner)
\end{aligned}$$

The product case is unsurprising: we represent a pair as a pointer to a heap-allocated record containing the sub-components. As discussed in section 2, other representations are possible.

We represent the ordinary lambda calculus unit as a pointer to the orderly lambda calculus unit. Recall that $|1| = 0$ in λ^{ord}. This means that our chosen representation of unit is as a pointer to a zero-word object. This corresponds precisely to the standard implementation of values of type unit as a distinguished pointer to nothing (e.g. the null pointer).

An analogous translation is defined at the term level. The interesting case is the translation of pairing, since pairs are the only terms requiring allocation. We begin by defining a λ^{ord} function **pair**.

$$\begin{aligned}
&\mathbf{pair} : \tau_1 \to \tau_2 \to !(\tau_1 \bullet \tau_2) \overset{\text{def}}{=} \\
&\quad \lambda(x_1 : \tau_1).\lambda(x_2 : \tau_2). \\
&\qquad \mathtt{reserve}_2\ \mathtt{as}\ a && (1) \\
&\qquad \mathtt{in}\ \mathtt{let}\ a_1 \bullet a_{2*} = a && (2) \\
&\qquad \mathtt{in}\ \mathtt{let}\ a_2 \bullet a_* = a_{2*} && (3) \\
&\qquad \mathtt{in}\ \mathtt{let} * = a_* && (4) \\
&\qquad \mathtt{in}\ a_1 := x_1\ \mathtt{as}\ a'_1 && (5) \\
&\qquad \mathtt{in}\ a_2 := x_2\ \mathtt{as}\ a'_2 && (6) \\
&\qquad \mathtt{in}\ \mathtt{alloc}(a'_1 \bullet a'_2)\ \mathtt{as}\ x && (7) \\
&\qquad \mathtt{in}\ \mathtt{ret}\ x && (8)
\end{aligned}$$

The first line of the function reserves the space on the frontier from which the pair will be created. This binds a single ordered variable a which points to the beginning of this space. Line 2 gives the names a_1 and a_{2*} respectively to the first and second words of the newly allocated space. From the typing rule for `reserve` we can see that the second location has an extra zero-byte value of type `unit` attached, so lines 3 and 4 serve to split out and eliminate this. Lines 5 and 6 initialize the two locations, renaming them to a'_1 and a'_2. Finally, line 7 allocates the initialized space into the heap and names the result x, which becomes the return value of the function in line 8.

This definition demonstrates how the various operations interact to permit low-level code to be written in a relatively high-level manner. In particular, there is no mention of offsets at all: everything is done in terms of standard alpha-varying variables. It may seem that this code is somewhat verbose, but it is simple to define syntactic abbreviations and composite terms that eliminate much of the verbosity. For example, in the common case for initialization terms where the coercion term Q is a variable, we may take advantage of alpha-conversion to simply re-use the old variable name, yielding a more standard looking assignment syntax.

$$a_1 := M\ \mathtt{in} E \quad \overset{\text{def}}{=} \quad a_1 := M\ \mathtt{as}\ a_1\ \mathtt{in} E$$

It is also trivial to define a composite reserve operation that precomputes the offset variables.

$$\frac{\Gamma; a_1{:}\mathtt{NS}, \ldots, a_n{:}\mathtt{NS} \vdash_{\text{exp}} E : \tau}{\Gamma; \Omega \vdash_{\text{exp}} \mathtt{reserve}_n\ \mathtt{as}[a_1, \ldots, a_n]\ \mathtt{in} E : \tau}$$

Working out the definition of this term is left as an exercise to the reader, but using these abbreviations, we can write the **pair** constructor quite succinctly.

$$\begin{aligned}
&\mathbf{pair} : \tau_1 \to \tau_2 \to !(\tau_1 \bullet \tau_2) \overset{\text{def}}{=} \\
&\quad \lambda(x_1 : \tau_1).\lambda(x_2 : \tau_2). \\
&\qquad \mathtt{reserve}_2\ \mathtt{as}[a_1, a_2] \\
&\qquad \mathtt{in}\ a_1 := x_1 \\
&\qquad \mathtt{in}\ a_2 := x_2 \\
&\qquad \mathtt{in}\ \mathtt{alloc}(a_1 \bullet a_2)\ \mathtt{as}\ x \\
&\qquad \mathtt{in}\ \mathtt{ret}\ x
\end{aligned}$$

The elimination forms for pairs can be given succinct definitions using the direct load defined in Figure 6.

$$\begin{array}{lcl}
\mathbf{fst} : !(\tau_1 \bullet \tau_2) \to \tau_1 & \overset{\text{def}}{=} & \lambda(x : !(\tau_1 \bullet \tau_2)) \\
 & & \quad \texttt{load}_{\tau_1 \bullet \tau_2}\ x_1 = x[0] \\
 & & \quad \texttt{in ret}\ x_1 \\
\mathbf{snd} : !(\tau_1 \bullet \tau_2) \to \tau_2 & \overset{\text{def}}{=} & \lambda(x : !(\tau_1 \bullet \tau_2)) \\
 & & \quad \texttt{load}_{\tau_1 \bullet \tau_2}\ x_2 = x[1] \\
 & & \quad \texttt{in ret}\ x_2
\end{array}$$

The remainder of the translation of the simply typed lambda calculus is straightforward. All variables introduced by the translation are assumed to be fresh.

$$\begin{array}{rl}
\ulcorner x \urcorner \overset{\text{def}}{=} & \texttt{ret}\ x \\
\ulcorner \overline{n} \urcorner \overset{\text{def}}{=} & \texttt{ret}\ \overline{n} \\
\ulcorner () \urcorner \overset{\text{def}}{=} & \texttt{ret}\ !* \\
\ulcorner \lambda(x{:}\tau).e \urcorner \overset{\text{def}}{=} & \texttt{ret}\ \lambda(x{:}\ulcorner \tau \urcorner).\ulcorner e \urcorner \\
\ulcorner e_1\, e_2 \urcorner \overset{\text{def}}{=} & \texttt{let}\ x_1 = \ulcorner e_1 \urcorner \\
 & \texttt{in let}\ x_2 = \ulcorner e_2 \urcorner \\
 & \texttt{in}\ x_1 x_2 \\
\ulcorner (e_1, e_2) \urcorner \overset{\text{def}}{=} & \texttt{let}\ x_1 = \ulcorner e_1 \urcorner \\
 & \texttt{in let}\ x_2 = \ulcorner e_2 \urcorner \\
 & \texttt{in let}\ x_t = \mathbf{pair}\, x_1 \\
 & \texttt{in let}\ x = x_t\, x_2 \\
 & \texttt{in ret}\ x \\
\ulcorner \pi_1 e \urcorner \overset{\text{def}}{=} & \texttt{let}\, x = \ulcorner e \urcorner \\
 & \texttt{in}\ \mathbf{fst}\, x \\
\ulcorner \pi_2 e \urcorner \overset{\text{def}}{=} & \texttt{let}\, x = \ulcorner e \urcorner \\
 & \texttt{in}\ \mathbf{snd}\, x
\end{array}$$

5.1 Coalescing reservation

Translating simply typed lambda calculus terms into the orderly lambda calculus breaks the high level memory abstractions and exposes a finer grain of detail. Exposing these details can enable optimizations not expressible at the more abstract level. A simple example of this is the ability to coalesce multiple calls to the allocator. For example, consider the result of translating the term $(3,(4,5))$ under the above translation (with some minor simplifications).

$$\begin{array}{l}
\ulcorner (3,(4,5)) \urcorner = \texttt{let}\, x_t = \texttt{reserve}_2\ \texttt{as}[a_1, a_2] \\
\qquad\qquad\qquad\qquad \texttt{in}\ a_1 := 4\, \texttt{as}\ a'_1 \\
\qquad\qquad\qquad\qquad \texttt{in}\ a_2 := 5\, \texttt{as}\ a'_2 \\
\qquad\qquad\qquad\qquad \texttt{in alloc}(a'_1 \bullet a'_2)\ \texttt{as}\ x \\
\qquad\qquad\qquad\qquad \texttt{in ret}\ x \\
\qquad\qquad \texttt{in reserve}_2\ \texttt{as}[a_3, a_4] \\
\qquad\qquad \texttt{in}\ a_3 := 3\, \texttt{as}\ a'_3 \\
\qquad\qquad \texttt{in}\ a_4 := x_t\, \texttt{as}\ a'_4 \\
\qquad\qquad \texttt{in alloc}(a'_3 \bullet a'_4)\ \texttt{as}\ x \\
\qquad\qquad \texttt{in ret}\ x
\end{array}$$

This code fragment makes two separate calls to the allocator, each reserving two words of space. It is easy to see that the second reserve operation can be *coalesced* with the first, reducing the total number of calls to the allocator.

$$\begin{array}{rl}
\ulcorner (3,(4,5)) \urcorner \overset{\text{opt}}{=} & \texttt{reserve}_4\ \texttt{as}[a_1, a_2, a_3, a_4] \\
 & \texttt{in}\ a_1 := 4\, \texttt{as}\ a'_1 \\
 & \texttt{in}\ a_2 := 5\, \texttt{as}\ a'_2 \\
 & \texttt{in alloc}(a'_1 \bullet a'_2)\ \texttt{as}\ x_t \\
 & \texttt{in}\ a_3 := 3\, \texttt{as}\ a'_3 \\
 & \texttt{in}\ a_4 := x_t\, \texttt{as}\ a'_4 \\
 & \texttt{in alloc}(a'_3 \bullet a'_4)\ \texttt{as}\ x \\
 & \texttt{in ret}\ x
\end{array}$$

This kind of optimization is commonly done in untyped compilers, but here we can easily express it in a typed setting.

A further step to consider is to try to coalesce the two allocation operations, in addition to coalescing the reservations. Unfortunately, this is not in general possible in our setting. The problem is that we currently cannot express pointers into the frontier—such pointers would be difficult to typecheck since the types of locations in the frontier can change. Therefore we are unable to initialize the second field of the top level pair until we have moved the other pair into the heap.

6 Extensions and future work

This paper has given a detailed presentation of the core of the orderly lambda calculus, developing a high-level framework for discussing issues of allocation and data-layout. The full language includes an account of sums and recursive types that permits sum allocation and tagging to be done using only the memory mechanisms already described. In addition, we have extended the coercion level to include ordered functions and application forms and shown that a rich language of coercions is definable in this setting. Finally, we have shown how the `reserve`, `alloc`, and `write` primitives can be replaced by typed constants, eliminating the need to incorporate special memory-management primitives into the language. The full language is described in a separate technical report [13].

The most important question that we have not yet addressed is how to give an account of the allocation of objects with dynamic extent. The system we have developed so far is predicated on the ability to statically predict the size of an object based on its type. For objects such as arrays however, this is clearly not true.

While an ad-hoc treatment of arrays can be fairly easily integrated into the language, this is unsatisfactory since the intention is to make all allocation explicit through the same mechanism. A more interesting possibility is to use a dependent type formalism [23] or a type analysis formalism [4] to introduce a notion of dynamic extent into the type system. We intend to explore this avenue further in the future.

Another important area for future research is to attempt to account for pointers into the frontier itself. As we saw in Section 5 we are forced to allocate an object into the heap before we can initialize other objects with a pointer to it, which prevents some useful optimizations such as the *destination passing style* optimization [8].

7 Related work

Ordered logic and ordered type theory have been explored extensively by Pfenning and Polakow [16, 15].

There is a significant amount of previous work applying ordinary linear type theory to memory management [1, 22, 5, 7], but none of it addresses (nor is intended to address) the question of separating out allocation and initialization, and of giving a foundational account of data layout.

The work that most closely addresses the issues that we discuss here is the alias type formalism of Smith, Walker, and Morrisett [20]). Alias types allow aliasing information to be tracked exactly in the type system. A quasi-linear type system allows memory locations to be destructively updated. Since aliasing is tracked exactly, an explicit "free" operation is provided which de-allocates space. Some very useful optimizations such as the destination passing style optimization can be encoded fairly easily in this language. The alias type formalism does not seem to provide for the explicit coalescing of allocator calls, nor does it provide an explicit type theory for describing data layout in the manner that we have attempted to do.

8 References

[1] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *Journal of Functional Programming*, 6(2):195–244, 1996.

[2] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'00)*, pages 95–107, Vancouver, Canada, June 2000. ACM Press.

[3] Karl Crary and Greg Morrisett. Type structure for low-level programming langauges. In *Twenty-Sixth International Colloquium on Automata, Languages, and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 40–54, Prague, Czech Republic, July 1999. Springer-Verlag.

[4] Karl Crary and Stephanie Weirich. Flexible type analysis. In *1999 ACM International Conference on Functional Programming*, Paris, France, September 1999. ACM Press.

[5] A. Igarashi and N. Kobayashi. Garbage collection based on a linear type system, 2000.

[6] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c, June 2002.

[7] Naoki Kobayashi. Quasi-linear types. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 29–42, New York, NY, 1999.

[8] Y. Minamide. A functional represention of data structures with a hole. In *Conference Record of the 25th Symposium on Principles of Programming Languages (POPL '98)*, 1998.

[9] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, Georgia, May 1999.

[10] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In A. Gordon and A. Pitts, editors, *Higer Order Operational Techniques in Semantics*. Newton Institute, Cambridge University Press, 1997.

[11] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[12] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 333–344, Montreal, Canada, June 1998. ACM Press.

[13] Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. A type theory for memory allocation and data layout. Technical Report CMU-CS-02-171, Department of Computer Science, Carnegie Mellon University, December 2002.

[14] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

[15] Jeff Polakow. *Ordered linear logic and applications*. PhD thesis, Carnegie Mellon University, June 2001. Available as Technical Report CMU-CS-01-152.

[16] Jeff Polakow and Frank Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In J.-Y. Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA'99)*, pages 295–309, L'Aquila, Italy, April 1999. Springer-Verlag LNCS 1581.

[17] Jeff Polakow and Frank Pfenning. Relating natural deduction and sequent calculus for intuitionistic non-commutative linear logic. In Andre Scedrov and Achim Jung, editors, *Proceedings of the 15th Conference on Mathematical Foundations of Programming Semantics*, New Orleans, Louisiana, April 1999. Electronic Notes in Theoretical Computer Science, Volume 20.

[18] Jeff Polakow and Frank Pfenning. Properties of terms in continuation-passing style in an ordered logical framework. In J.Despeyroux, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA'99)*, Santa Barbara, California, June 2000.

[19] Zhong Shao. An overview of the FLINT/ML compiler. In *1997 Workshop on Types in Compilation*, Amsterdam, June 1997. ACM SIGPLAN. Published as Boston College Computer Science Department Technical Report BCCS-97-03.

[20] Frederick Smith, David Walker, and Greg Morrisett. Alias types. *Lecture Notes in Computer Science*, 1782, 2000.

[21] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996.

[22] David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227(1–2):231–248, 1999.

[23] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 249–257, Montreal, Canada, June 1998. ACM Press.

A Static semantics

Definitions

$$\begin{array}{rcl} \tau^0 &=& 1 \\ \tau^{n+1} &=& \tau \bullet \tau^n \end{array} \qquad (\Omega_1,\Omega_2) \overset{\text{def}}{=} \begin{cases} \Omega_2 & \text{if } \Omega_1 = \cdot \\ a{:}\tau,(\Omega_1',\Omega_2) & \text{if } \Omega_1 = a{:}\tau,\Omega_1' \end{cases}$$

Well-formed contexts and frontier $\boxed{\vdash \Gamma\ ,\ \vdash \Omega\ ,\ \vdash \omega : \Omega}$

$$\frac{}{\vdash \cdot} \qquad \frac{\vdash \tau : \mathtt{T}_{\mathrm{reg}} \quad \vdash \Gamma \quad (x \notin \Gamma)}{\vdash x{:}\tau,\Gamma} \qquad \frac{}{\vdash \cdot} \qquad \frac{\vdash \Omega \quad (a \notin \Omega)}{\vdash a{:}\tau,\Omega} \qquad \frac{}{\vdash \cdot : \cdot} \qquad \frac{\vdash_{\mathrm{val}} V : \tau \quad \vdash \omega : \Omega \quad (a \notin \Omega)}{\vdash a \mapsto V, \omega : a{:}\tau,\Omega}$$

Register and heap types $\boxed{\vdash \tau : \mathtt{T}_{reg},\ \vdash \tau : \mathtt{T}_{h}}$

$$\frac{}{\vdash \mathtt{int} : \mathtt{T}_{\mathrm{reg}}} \qquad \frac{}{\vdash \mathtt{NS} : \mathtt{T}_{\mathrm{reg}}} \qquad \frac{\vdash \tau : \mathtt{T}_{\mathrm{h}}}{\vdash\ !\tau : \mathtt{T}_{\mathrm{reg}}} \qquad \frac{\vdash \tau_1 : \mathtt{T}_{\mathrm{reg}} \quad \vdash \tau_2 : \mathtt{T}_{\mathrm{reg}}}{\vdash \tau_1 \to \tau_2 : \mathtt{T}_{\mathrm{reg}}} \qquad \frac{}{\vdash 1 : \mathtt{T}_{\mathrm{h}}} \qquad \frac{\vdash \tau_1 : \mathtt{T}_{\mathrm{h}} \quad \vdash \tau_2 : \mathtt{T}_{\mathrm{h}}}{\vdash \tau_1 \bullet \tau_2 : \mathtt{T}_{\mathrm{h}}} \qquad \frac{\vdash \tau : \mathtt{T}_{\mathrm{reg}}}{\vdash \tau : \mathtt{T}_{\mathrm{h}}}$$

Coercion terms $\boxed{\Omega \vdash_{crc} Q : \tau}$

$$\frac{}{a{:}\tau \vdash_{\mathrm{crc}} a : \tau} \qquad \frac{}{\cdot \vdash_{\mathrm{crc}} * : 1} \qquad \frac{\Omega_1 \vdash_{\mathrm{crc}} Q_1 : \tau_1 \quad \Omega_2 \vdash_{\mathrm{crc}} Q_2 : \tau_2}{\Omega_1,\Omega_2 \vdash_{\mathrm{crc}} Q_1 \bullet Q_2 : \tau_1 \bullet \tau_2}$$

Terms $\boxed{\Gamma;\Omega \vdash_{trm} M : \tau}$

$$\frac{\Gamma(x) = \tau}{\Gamma;\cdot \vdash_{\mathrm{trm}} x : \tau} \qquad \frac{}{\Gamma;\cdot \vdash_{\mathrm{trm}} \overline{n} : \mathtt{int}} \qquad \frac{}{\Gamma;\cdot \vdash_{\mathrm{trm}} \mathtt{ns} : \mathtt{NS}} \qquad \frac{\vdash_{\mathrm{val}} V : \tau}{\Gamma;\cdot \vdash_{\mathrm{trm}} !V : !\tau} \qquad \frac{\vdash \tau : \mathtt{T}_{\mathrm{reg}} \quad \Gamma,x{:}\tau;\Omega \vdash_{\mathrm{exp}} E : \tau'}{\Gamma;\Omega \vdash_{\mathrm{trm}} \lambda(x{:}\tau).E : \tau \to \tau'}$$

Values $\boxed{\vdash_{val} V : \tau}$

$$\frac{}{\vdash_{\mathrm{val}} \overline{n} : \mathtt{int}} \qquad \frac{}{\vdash_{\mathrm{val}} \mathtt{ns} : \mathtt{NS}} \qquad \frac{\vdash_{\mathrm{val}} V : \tau}{\vdash_{\mathrm{val}} !V : !\tau} \qquad \frac{\vdash \tau : \mathtt{T}_{\mathrm{reg}} \quad x{:}\tau;\cdot \vdash_{\mathrm{exp}} E : \tau'}{\vdash_{\mathrm{val}} \lambda(x{:}\tau).E : \tau \to \tau'} \qquad \frac{}{\vdash_{\mathrm{val}} * : 1} \qquad \frac{\vdash_{\mathrm{val}} V_1 : \tau_1 \quad \vdash_{\mathrm{val}} V_2 : \tau_2}{\vdash_{\mathrm{val}} V_1 \bullet V_2 : \tau_1 \bullet \tau_2}$$

Expressions $\boxed{\Gamma;\Omega \vdash_{exp} E : \tau}$

$$\frac{\Gamma;\cdot \vdash_{\mathrm{trm}} M : \tau}{\Gamma;\cdot \vdash_{\mathrm{exp}} \mathtt{ret}\, M : \tau} \qquad \frac{\Gamma;\Omega \vdash_{\mathrm{trm}} M_1 : \tau_1 \to \tau_2 \quad \Gamma;\cdot \vdash_{\mathrm{trm}} M_2 : \tau_1}{\Gamma;\Omega \vdash_{\mathrm{exp}} M_1\, M_2 : \tau_2} \qquad \frac{\Gamma;\Omega \vdash_{\mathrm{exp}} E_1 : \tau_1 \quad \Gamma,x{:}\tau_1;\cdot \vdash_{\mathrm{exp}} E_2 : \tau_2}{\Gamma;\Omega \vdash_{\mathrm{exp}} \mathtt{let}\, x{:}\tau_1 = E_1\, \mathtt{in}\, E_2 : \tau_2}$$

$$\frac{\Gamma;a{:}\mathtt{NS}^n \vdash_{\mathrm{exp}} E : \tau}{\Gamma;\Omega \vdash_{\mathrm{exp}} \mathtt{reserve}_n\, \mathtt{as}\, a\, \mathtt{in}\, E : \tau} \qquad \frac{\Omega_L \vdash_{\mathrm{crc}} Q : \tau \quad \Gamma,x{:}!\tau;\Omega_R \vdash_{\mathrm{exp}} E : \tau'}{\Gamma;\Omega_L,\Omega_R \vdash_{\mathrm{exp}} \mathtt{alloc}\, Q\, \mathtt{as}\, x\, \mathtt{in}\, E : \tau'}$$

$$\frac{\Omega \vdash_{\mathrm{crc}} Q : \tau \quad \vdash \tau : \mathtt{T}_{\mathrm{reg}} \quad \Gamma;\cdot \vdash_{\mathrm{trm}} M : \tau' \quad \Gamma;\Omega_L,a{:}\tau',\Omega_R \vdash_{\mathrm{exp}} E : \tau''}{\Gamma;\Omega_L,\Omega,\Omega_R \vdash_{\mathrm{exp}} Q := M\, \mathtt{as}\, a\, \mathtt{in}\, E : \tau''} \qquad \frac{\Omega \vdash_{\mathrm{crc}} Q : 1 \quad \Gamma;\Omega_L,\Omega_R \vdash_{\mathrm{exp}} E : \tau}{\Gamma;\Omega_L,\Omega,\Omega_R \vdash_{\mathrm{exp}} \mathtt{let}\, * = Q\, \mathtt{in}\, E : \tau}$$

$$\frac{\Omega \vdash_{\text{crc}} Q : \tau \quad \Gamma;\Omega_1, a{:}\tau, \Omega_2 \vdash_{\text{exp}} E : \tau'}{\Gamma;\Omega_1,\Omega,\Omega_2 \vdash_{\text{exp}} \mathtt{let}\, a = Q\, \mathtt{in}\, E : \tau'} \qquad \frac{\Omega \vdash_{\text{crc}} Q : \tau_1 \bullet \tau_2 \quad \Gamma;\Omega_1, a_1{:}\tau_1, a_2{:}\tau_2, \Omega_2 \vdash_{\text{exp}} E : \tau}{\Gamma;\Omega_1,\Omega,\Omega_2 \vdash_{\text{exp}} \mathtt{let}\, a_1 \bullet a_2 = Q\, \mathtt{in}\, E : \tau}$$

$$\frac{\Gamma;\cdot \vdash_{\text{trm}} M : !(\tau_1 \bullet \tau_2) \quad \Gamma, x_1{:}!\tau_1, x_2{:}!\tau_2;\Omega \vdash_{\text{exp}} E : \tau}{\Gamma;\Omega \vdash_{\text{exp}} \mathtt{let}\,!x_1 \bullet x_2 = M\, \mathtt{in}\, E : \tau} \qquad \frac{\Gamma;\cdot \vdash_{\text{trm}} M : !\tau_1 \quad \vdash \tau_1 : \mathrm{T}_{\text{reg}} \quad \Gamma, x{:}\tau_1,;\Omega \vdash_{\text{exp}} E : \tau_2}{\Gamma;\Omega \vdash_{\text{exp}} \mathtt{let}\,!x = M\, \mathtt{in}\, E : \tau_2}$$

B Dynamic semantics

Definitions

$$\begin{array}{lcl} V^0 & = & * \\ V^{n+1} & = & V \bullet V^n \end{array} \qquad (\omega_1,\omega_2) \overset{\text{def}}{=} \begin{cases} \omega_2 & \text{if } \omega_1 = \cdot \\ a \mapsto V, (\omega_1', \omega_2) & \text{if } \omega_1 = a \mapsto V, \omega_1' \end{cases} \qquad \begin{array}{lcl} *[\cdot] & = & * \\ a[a \mapsto V] & = & V \\ (Q_1 \bullet Q_2)[\omega_1,\omega_2] & = & Q_1[\omega_1] \bullet Q_1[\omega_2] \end{array}$$

Expressions

$$\boxed{(\omega, E) \mapsto (\omega', E')}$$

$$\frac{}{(\omega, (\lambda(x{:}\tau).E)\, M_v) \mapsto (\omega, E[M_v/x])}$$

$$\frac{(\omega, E_1) \mapsto (\omega', E_1')}{(\omega, \mathtt{let}\, x{:}\tau = E_1\, \mathtt{in}\, E_2) \mapsto (\omega', \mathtt{let}\, x{:}\tau = E_1'\, \mathtt{in}\, E_2)} \qquad \frac{}{(\cdot, \mathtt{let}\, x{:}\tau = \mathtt{ret}\, M_v\, \mathtt{in}\, E) \mapsto (\cdot, E[M_v/x])}$$

$$\frac{}{(\omega, \mathtt{reserve}_n\, \mathtt{as}\, a\, \mathtt{in}\, E) \mapsto (a \mapsto \mathtt{ns}^n, E)} \qquad \frac{Q_v[\omega_1] = V}{((\omega_1,\omega_2), \mathtt{alloc}\, Q_v\, \mathtt{as}\, x\, \mathtt{in}\, E \mapsto (\omega_2, E[!V/x])}$$

$$\frac{}{((\omega_1, a \mapsto V, \omega_2), a := M_v\, \mathtt{as}\, a'\, \mathtt{in}\, E) \mapsto ((\omega_1, a' \mapsto M_v, \omega_2), E)}$$

$$\frac{}{(\omega, \mathtt{let}\, a = Q_v\, \mathtt{in}\, E) \mapsto (\omega, E[Q_v/a])}$$

$$\frac{}{(\omega, \mathtt{let}\, a_1 \bullet a_2 = Q_1 \bullet Q_2\, \mathtt{in}\, E) \mapsto (\omega, E[Q_1, Q_2/a_1, a_2])}$$

$$\frac{}{((\omega_1, a \mapsto V_1 \bullet V_2, \omega_2), \mathtt{let}\, a_1 \bullet a_2 = a\, \mathtt{in}\, E) \mapsto ((\omega_1, a_1 \mapsto V_1, a_2 \mapsto V_2, \omega_2), E)}$$

$$\frac{}{(\omega, \mathtt{let}\, * = *\, \mathtt{in}\, E) \mapsto (\omega, E)} \qquad \frac{}{((\omega_1, a \mapsto *, \omega_2), \mathtt{let}\, * = a\, \mathtt{in}\, E) \mapsto ((\omega_1,\omega_2), E)}$$

$$\frac{}{(\omega, \mathtt{let}\,!x_1 \bullet x_2 = !(V_1 \bullet V_2)\, \mathtt{in}\, E) \mapsto (\omega, E[!V_1, !V_2/x_1, x_2])} \qquad \frac{}{(\omega, \mathtt{let}\,!x = !V\, \mathtt{in}\, E) \mapsto (\omega, E[V/x])}$$