A Type Theory for Memory Allocation and Data Layout (Extended Version)

Leaf Petersen Robert Harper Karl Crary Frank Pfenning

December, 2002
CMU-CS-02-171

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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.

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.

Keywords: typed compilation, ordered type theory, memory management, garbage collection, data layout

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 [14, 22, 13] 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 [14, 22, 11, 3, 20, 7]. 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[13], 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 addition the type and size of the value to be initialized must be fixed when the space for the object is initially reserved.

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 flexible 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 $\mathtt{int} \times (\mathtt{int} \times \mathtt{int})$ 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.

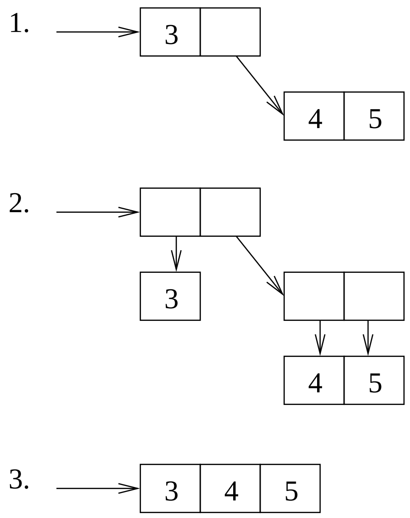

Figure 1: Three possible layouts for the term $(3,(4,5))$

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 [4].

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

Different memory-management systems combine these stages in different ways. In the TAL framework [13], reservation and allocation are done atomically, and hence initialization is very restricted in how it can change the type. In the alias type framework [24] primitives are provided for reservation and initialization, but the inability to express *may-alias* constraints may be seen as a lack of an allocation operation.

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

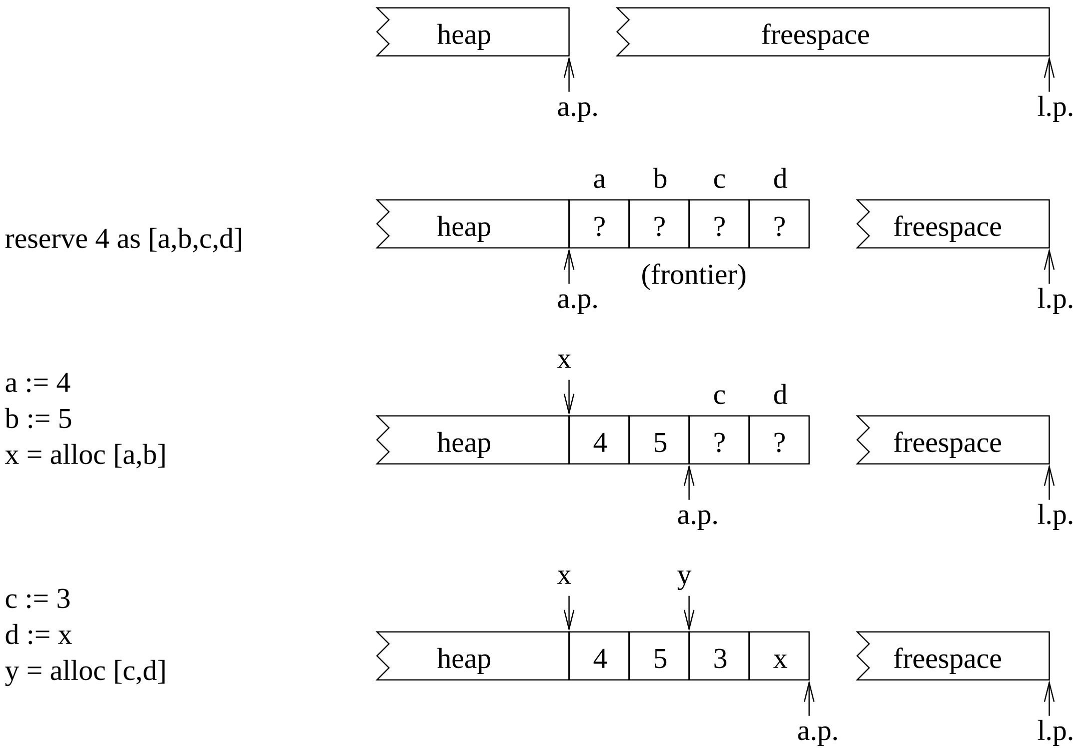

Figure 2: Reservation, initialization, and allocation of $(3,(4,5))$

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 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.

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 [18, 17, 19, 16]. 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.

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

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.

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

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

Figure 3: Standard ordered lambda calculus syntax

The left ordered lambda rules similarly do not permit hypotheses to move in the context.

$$
\frac{\Gamma; a{:}\tau_1, \Omega \vdash M : \tau_2}{\Gamma; \Omega \vdash \lambda^{<}(a{:}\tau_1).M : \tau_1 \rightarrowtail \tau_2}
\qquad
\frac{\Gamma; \Omega_2 \vdash M_1 : \tau_1 \rightarrowtail \tau_2 \quad \Gamma; \Omega_1 \vdash M_2 : \tau_2}{\Gamma; \Omega_1, \Omega_2 \vdash M_1 \,^{<}\, M_2 : \tau_2}
$$

Notice that left λ-abstraction typechecks its body with the argument on the left side of the ordered context. Therefore, the application form must split the context such that the argument draws its ordered resources from the left half. In this way, the evaluation of the application will preserve the order of the resources.

The right ordered lambda is analogous, but takes its arguments from the right side of the context instead of the left.

$$
\frac{\Gamma; \Omega, a{:}\tau_1 \vdash M : \tau_2}{\Gamma; \Omega \vdash \lambda^{>}(a{:}\tau_1).M : \tau_1 \twoheadrightarrow \tau_2}
\qquad
\frac{\Gamma; \Omega_1 \vdash M_1 : \tau_1 \twoheadrightarrow \tau_2 \quad \Gamma; \Omega_2 \vdash M_2 : \tau_2}{\Gamma; \Omega_1, \Omega_2 \vdash M_1 \,^{>}\, M_2 : \tau_2}
$$

Unrestricted functions may refer to ordered variables, but their arguments must be closed to prevent duplication of linear terms.

$$
\frac{\Gamma, x{:}\tau; \Omega \vdash M : \tau'}{\Gamma; \Omega \vdash \lambda(x{:}\tau).M : \tau \to \tau'}
\qquad
\frac{\Gamma; \Omega \vdash M_1 : \tau_1 \to \tau_2 \quad \Gamma; \cdot \vdash M_2 : \tau_1}{\Gamma; \Omega \vdash M_1\, M_2 : \tau_2}
$$

Finally, the mobility modal permits terms which 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 function that splits apart an ordered pair and reforms it in the opposite order is not well-typed.

$$\lambda^{<}(a{:}\tau_1 \bullet \tau_2).\,\texttt{let}\, a_1 \bullet a_2 = a\, \texttt{in}\, a_2 \bullet a_1$$

   Viewed as a linear (rather than ordered) term, this function 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 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 typechecking 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

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

Figure 4: Syntax

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 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. To simplify the presentation, this section will focus on a core language that captures the essential ideas. Some extensions to this core language (such as recursive types and sums) will be considered in Section 6.

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 $\mathtt{T}_{\mathrm{reg}}$ classifies the types of values which may be loaded into registers, whereas the kind $\mathtt{T}_{\mathrm{h}}$ classifies types that may be heap-allocated (a strict super-set of the former).

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}

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

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

For simplicity, the smallest unit of size we consider is a single machine word, and we assume that functions are unit size (which would be made explicit if we chose to account for closure conversion). The multiplicative unit type has size zero, since it is inhabited by a single predictable value which therefore does not need to be represented.

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.

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

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 [15], but without an explicit modal type for computations. It is likely that by using a full computational (or monadic) type system we could give a more general system that does not constrain evaluation order, but for our purposes we simply assume call-by-value evaluation.

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.

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

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

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

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 [12]). 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 may 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.

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, $\vdash_{\mathrm{val}} V : \tau$, 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" [9] 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 $|\Omega| = |\tau|$.

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

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 which we will consider in order.

Ordinary expressions

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

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

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 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.

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

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 sub-expression. 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.

$$\frac{\Gamma; \cdot \vdash_{\text{trm}} M : !\tau \quad \vdash \tau[i] : \mathtt{T}_{\text{reg}} \quad \Gamma, x{:}\tau[i]; \Omega \vdash_{\text{exp}} E : \tau_2}{\Gamma;\Omega \vdash_{\text{exp}} \mathtt{load}_\tau\, x = M[i] \,\mathtt{in}\, E : \tau_2}$$

$$\tau[i] \stackrel{\text{def}}{=} \begin{cases} \tau_1[i] & \text{if } \tau = \tau_1 \bullet \tau_2 \text{ and } |\tau_1| > i \\ \tau_2[i - |\tau_1|] & \text{if } \tau = \tau_1 \bullet \tau_2 \text{ and } |\tau_1| \leq i \\ \tau & \text{if } \tau \text{ is not a fuse and } i = 0 \end{cases}$$

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

Figure 6: An example direct-load defined in terms of locative-load.

Memory expressions

The most interesting and non-standard expressions are those dealing directly with the frontier. Recall that there are three operations in 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 bytes 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.

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

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 let. 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.

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

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. In practice, these operations correspond to offset calculations. For simple types (such as those presented here) such offset calculations can always be done statically. This means that we may view these expressions as coercions.

The simplest coercion expression is the elimination form for unit.

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

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.

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

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.

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

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.

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

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.

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

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 the use of locatives to give an efficient implementation. While not completely out of the question, locatives 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 locative computations before we arrive at a term that can be loaded directly. This is clearly impractical.

We choose to use this "locative-load" as the primitive notion because it provides a simple and well-motivated 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 the locative-load (figure 6). By taking such a direct-load as primitive and giving it a direct implementation, the need for the locatives 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 (given in Appendix C.2). ∎

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.

As we saw in Section 2, there are many different choices of representations for the high level notion of pairing. The translation to the orderly lambda calculus makes these representation choices explicit in the types. We therefore define a translation $\ulcorner\tau\urcorner$ that maps each ordinary lambda calculus type to a type in the orderly lambda calculus as follows.

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

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).

At the term level, every simply typed lambda calculus term is translated to an orderly lambda calculus expression. The interesting case is the translation of pairing, since pairs are the only terms requiring allocation. To help with the translation, we define a function **pair** that takes two terms, allocates a two-word area in memory, initializes the memory with the terms, and returns a pointer to the newly created cell.

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

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, and then line 7 allocates the initialized space into the heap and names the result x.

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 fairly simple to define composite terms that eliminate much of the verbosity. For example, it is trivial to define a composite reserve operation that pre-computes the offset variables.

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

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

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

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

$$\begin{aligned}
\textbf{fst} : !(\tau_1 \bullet \tau_2) \to \tau_1 \quad &\stackrel{\text{def}}{=} \quad \lambda(x : !(\tau_1 \bullet \tau_2)) \\
&\qquad\quad \texttt{load}\, x_1 = x[0] \\
&\qquad\quad \texttt{in}\,\texttt{ret}\, x_1 \\
\textbf{snd} : !(\tau_1 \bullet \tau_2) \to \tau_2 \quad &\stackrel{\text{def}}{=} \quad \lambda(x : !(\tau_1 \bullet \tau_2)) \\
&\qquad\quad \texttt{load}\, x_2 = x[1] \\
&\qquad\quad \texttt{in}\,\texttt{ret}\, x_2
\end{aligned}$$

As an exercise, the reader may wish to write the definitions of **fst** and **snd** in terms of the primitive locative operations and verify that it is equivalent to the definitional expansion of the direct load version.

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

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

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).

$$
\begin{aligned}
\ulcorner (3,(4,5)) \urcorner = \texttt{let}\, x_t = {} & \texttt{reserve}_2\, \texttt{as}[a_1, a_2] \\
& \texttt{in}\, a_1 := 4\ \texttt{as}\ a_1' \\
& \texttt{in}\, a_2 := 5\ \texttt{as}\ a_2' \\
& \texttt{in}\, \texttt{alloc}(a_1' \bullet a_2')\ \texttt{as}\ x \\
& \texttt{in}\, \texttt{ret}\, x \\
\texttt{in}\, \texttt{reserve}_2\, & \texttt{as}[a_3, a_4] \\
\texttt{in}\, a_3 := 3\ & \texttt{as}\ a_3' \\
\texttt{in}\, a_4 := x_t\ & \texttt{as}\ a_4' \\
\texttt{in}\, \texttt{alloc}(a_3' & \bullet a_4')\ \texttt{as}\ x \\
\texttt{in}\, \texttt{ret}\, x &
\end{aligned}
$$

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.

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

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 do not currently have a way to 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. While this is insignificant here, there are situations in which the ability to coalesce allocates would be valuable. For example, when allocating large lists we only care about getting a pointer to the first element. It would be preferable to be able to reserve and initialize the entire list in the frontier, and then perform a single alloc instruction to get a pointer to the last element.

6 Extensions and future work

The previous section gave a detailed presentation of the core language of the orderly lambda calculus, developing a framework for discussing issues of allocation and data-layout in a lambda calculus setting. This section discusses at a more informal level a number of useful extensions to the core language that could be developed in more detail in a full length paper.

6.1 Sum types

A basic but interesting extension to consider is how to give an account of sum types. Just as was the case with the pair type, the lambda calculus level sum type $\tau_1 + \tau_2$ is a high level abstraction that avoids dealing with the particular issues of how the two branches of the sum are to be discriminated. At a low level, a value of sum type might correspond to a pointer to a record containing a tag indicating which branch of the sum was occupied and the value itself. Creating such a value entails the allocation and initialization of two separate words of memory. While it is possible to use ad-hoc mechanisms to account for this, we would prefer to give a foundational account of sum types in the same way that we have done for pairs.

To make this possible, we begin by definining a new judgement $\vdash \tau_1 \# \tau_2$ which captures the idea of types whose inhabitants can always be distinguished at runtime. If $\vdash \tau_1 \# \tau_2$ then it must be the case that the set of runtime values classified by τ_1 is disjoint from the set of runtime values classified by τ_2. The separation of this judgement makes the rest of the type theory parametric over a theory of discriminability: the question of *how* values are discriminated is separated from the question of *what* values are discriminable.

This judgement is internalized into the type system via a new type $\tau_1 \oplus \tau_2$ which corresponds to the binary union of two types which are known to be discriminable.

$$\frac{\vdash \tau_1 : \mathtt{T}_{\mathrm{reg}} \qquad \vdash \tau_2 : \mathtt{T}_{\mathrm{reg}} \qquad \vdash \tau_1 \# \tau_2}{\vdash \tau_1 \oplus \tau_2 : \mathtt{T}_{\mathrm{reg}}}$$

For simplicity, we insist that both branches of the disjoint sum be of word size: a more general system might permit unions of more general types.

The specification of exactly what types are discriminable is given by the definition of the new disjointness judgement. At the very least, we require that the judgement be symmetric, and distributive over disjoint unions.

$$\frac{\vdash \tau_2 \# \tau_1}{\vdash \tau_1 \# \tau_2} \qquad \frac{\vdash \tau \# \tau_1 \qquad \vdash \tau \# \tau_2}{\vdash \tau \# (\tau_1 \oplus \tau_2)}$$

This then provides a starting point for our treatment of sums. In the style of the TALx86 implementation [11], we add a family of types $\mathtt{Tag}_i$ classifying tag values $\mathtt{tag}_i$ for integer indices i.

$$\frac{}{\vdash \mathtt{Tag}_i : \mathtt{T}_{\mathrm{reg}}} \qquad \frac{}{\Gamma; \cdot \vdash \mathtt{tag}_i : \mathtt{Tag}_i}$$

The intention is that these values be used as the basis for discriminating the arms of sums. So for example, a simple un-optimized translation of the lambda calculus sum type would given as follows.

$$\ulcorner \tau_1 + \tau_2 \urcorner = !(\mathtt{Tag}_0 \bullet \ulcorner \tau_1 \urcorner) \oplus !(\mathtt{Tag}_1 \bullet \ulcorner \tau_2 \urcorner)$$

We incorporate the fact that we can distinguish tagged pairs into the type system by adding a rule to the discriminability judgement.

$$\frac{(i \neq j)}{\vdash\ !(\texttt{Tag}_i \bullet \tau) \#\ !(\texttt{Tag}'_j \bullet \tau')}$$

A more general strategy might be to permit tags to be located anywhere.

$$\frac{(i \neq j)}{\vdash \texttt{Tag}_i \# \texttt{Tag}_j} \qquad \frac{\vdash \tau_1 \# \tau'_1}{\vdash \tau_1 \bullet \tau_2 \# \tau'_1 \bullet \tau'_2} \qquad \frac{\vdash \tau_2 \# \tau'_2}{\vdash \tau_1 \bullet \tau_2 \# \tau'_1 \bullet \tau'_2}$$

In practice however, this level of generality is never used.

An optimization that is commonly performed on sum types is to observe that pointers are always discriminable from small tags: therefore we can sometimes avoid tagging values. This fact can be expressed succinctly by adding a discrimination rule that says that pointers are discriminable from small tags (where small is arbitrarily chosen to mean less than 256).

$$\frac{(i < 256)}{\vdash \texttt{Tag}_i \#\ !\tau}$$

Introducing terms of the disjoint sum type requires two additional constructs at the term (and value) level.

$$\frac{\Gamma; \Omega \vdash_{\text{trm}} M : \tau \quad \vdash \tau' \# \tau}{\Gamma; \Omega \vdash_{\text{trm}} \texttt{inl}_{\tau'}\, M : \tau \oplus \tau'} \qquad \frac{\Gamma; \Omega \vdash_{\text{trm}} M : \tau' \quad \vdash \tau \# \tau'}{\Gamma; \Omega \vdash_{\text{trm}} \texttt{inr}_{\tau}\, M : \tau \oplus \tau'}$$

The important point here is that since the types in the disjoint sum are required to be discriminable *before* they are injected into the sum, the tagging constructs here serve solely as injections into the sum type and do not have any runtime effect. In particular, we do *not* discriminate based on the `inl` or `inr` tag.

We eliminate disjoint sums with a case expression.

$$\frac{\begin{array}{c}\Gamma; \cdot \vdash_{\text{trm}} M : \tau_1 \oplus \tau_2 \\ \Gamma, x_1{:}\tau_1; \Omega \vdash_{\text{exp}} E_1 : \tau \quad \Gamma, x_2{:}\tau_2; \Omega \vdash_{\text{exp}} E_2 : \tau\end{array}}{\Gamma; \Omega \vdash_{\text{exp}} \texttt{case}\, M\, \texttt{of}\, (\texttt{inl}\, x_1 \Rightarrow E_1 \mid \texttt{inr}\, x_2 \Rightarrow E_2) : \tau}$$

The case construct abstracts the actual operational manner in which values of the two types are discriminated. We rely on the definition of the discriminability judgement to ensure that values of the types τ_1 and τ_2 can actually be discriminated during evaluation.

6.2 Recursive types

The question of recursive types is always an interesting one. In our setting, recursive types are particularly interesting because it is not immediately clear how to give a definition for the size function for such types, nor at what level the constructors and destructors should exist. There are a number of different approaches that can be taken, some giving significantly more expressive power than others. For the purposes of this presentation, we give a very simple interpretation which captures the manner in which recursive types are usually used.

The approach we shall take to add a recursive types constructor to the term level, and a destructor to the expression level.

$$\begin{array}{rcl} \tau & ::= & \ldots \mid \alpha \mid \mu(\alpha).\tau \\ M & ::= & \ldots \mid \texttt{roll}_\tau\, M \\ E & ::= & \texttt{let}\, \texttt{roll}\, x = M\, \texttt{in}\, E \end{array}$$

Since all terms are intended to fit into registers, we must therefore ensure that the $|\mu(\alpha).\tau| = 1$, at least for well-formed types. We enforce this property via the kinding rules for recursive types and variables.

$$\frac{}{\Delta, \alpha, \Delta' \vdash \alpha : \mathtt{T}_{\mathrm{h}}} \qquad \frac{\Delta, \alpha \vdash \tau : \mathtt{T}_{\mathrm{reg}}}{\Delta \vdash \mu(\alpha).\tau : \mathtt{T}_{\mathrm{reg}}}$$

Kinding rules are extended with type contexts Δ that list the available free variables. Recursive variables are always deemed to have the general $\mathtt{T}_{\mathrm{h}}$ kind. This enforces the property that any occurrences of the variable must be guarded under a ! type. It is easy to show that if $\Delta \vdash \mu(\alpha).\tau : \mathtt{T}_{\mathrm{reg}}$ then $|\mu(\alpha).\tau| = |\tau| = 1$, where $|\alpha|$ is undefined. For example, we may verify that the type of lists $\mu(\alpha).\,!((\mathtt{int} \bullet \alpha) + 1)$ has size one.

$$|\mu(\alpha).\,!((\mathtt{int} \bullet \alpha) + 1)| = |\,!((\mathtt{int} \bullet \alpha) + 1)| = 1$$

This account of recursive types is sufficient for most practical purposes, but it does forbid some apparently reasonable types. For example, an alternative implementation of lists that seems equally reasonable is to place the indirection outside of the recursive type.

$$(!\,\mu(\alpha).(\mathtt{int} \bullet (!\,\alpha)) + 1)$$

This is not permitted by this simple account of recursive types, since the type of the body is a fuse and hence does not have kind $\mathtt{T}_{\mathrm{reg}}$.

More general theories of recursive types can be formulated by incorporating a richer kind structure that makes finer distinction between types of various sizes.

6.3 Coercions

An obvious extension to consider is the addition of the ordered function types to the language. Recall the two ordered function types from Section 3: $\tau_1 \rightarrowtail \tau_2$ taking its argument from the left side of the ordered context, and $\tau_1 \twoheadrightarrow \tau_2$ taking its argument from the right. What should be the interpretation of these types in the orderly lambda calculus? An interesting and useful idea is to consider these arrows as describing coercions that reorganize a section of memory. With this in mind, we place the ordered lambdas at the level of syntactic coercions.

$$\begin{array}{rcl} \tau & ::= & \ldots \mid \tau_1 \rightarrowtail \tau_2 \mid \tau_1 \twoheadrightarrow \tau_2 \\ Q & ::= & \ldots \mid \lambda^{<}(a{:}\tau).Q \mid Q_1 \;^{<}\; Q_2 \\ & & \mid \lambda^{>}(a{:}\tau).Q \mid Q_1 \;^{>}\; Q_2 \end{array}$$

The typing rules for these terms are exactly as given for the standard ordered lambda calculus (Section 3).

An important property of the orderly lambda calculus is that the size of a term in memory is uniquely given by its type. What then is the size of an ordered lambda? Given the interpretation of these functions as static coercions which simply re-organize our view of memory, it would seem that the size of these terms should be zero, as was the case with the ordered unit, and indeed for closed lambdas, this will be the case. However, this is not in general true, as the following typing derivation shows.

$$a_1{:}\mathtt{int} \vdash_{\mathrm{crc}} \lambda^{>}(a_2{:}\mathtt{int}).a_1 \bullet a_2 : \mathtt{int} \twoheadrightarrow \mathtt{int} \bullet \mathtt{int}$$

This term is clearly well-typed: however, it does not satisfy the size-preservation property. The size of the given context is one, but we have posited that the size of an ordered function type is zero: $|a_1{:}\mathtt{int}| = 1 \neq |\mathtt{int} \twoheadrightarrow \mathtt{int} \bullet \mathtt{int}| = 0$.

The problem is that the term given above represents an open coercion: it takes the right side of the frontier as an argument, but refers to the left side via a free variable. This is reflected in the type itself: the return type of the function is larger than the argument type: the extra space comes from free variables pointing into the frontier. This function can therefore be thought of as having part of its return value already allocated in the frontier. The other part of the return value will be provided as an argument. Therefore, the space occupied by such a function is the size of the types of its free variables. Since every free ordered variable must be used in the body, and since coercion terms are size preserving, this is in turn given by the type as follows.

$$\begin{array}{rcl} |\tau_1 \rightarrowtail \tau_2| & = & |\tau_2| - |\tau_1| \\ |\tau_1 \twoheadrightarrow \tau_2| & = & |\tau_2| - |\tau_1| \end{array}$$

The size of an ordered function is the size of its return type, less whatever space it abstracts over.

A simple example of a coercion function we might wish to write is the following, which introduces a zero-word unit value onto the front of a term.

$$\mathbf{luniti}_{[\tau]} : \tau_1 \rightarrowtail 1 \bullet \tau \quad \stackrel{\text{def}}{=} \quad \lambda^{<}(a : \tau).(* \bullet a)$$

However, there are very few interesting coercions we can write without adding elimination forms to the coercion language. This is not hard to do.

$$Q \quad ::= \quad \ldots \mid \mathtt{let}\, * = Q\, \mathtt{in}\, Q \mid \mathtt{let}\, a_1 \bullet a_2 = Q\, \mathtt{in}\, Q$$

The typing rules for these coercions are again exactly as given in Section 3.

With the addition of these terms, more coercions can be written. A simple example is the inverse of the previous coercion, which strips of a leading unit value.

$$\begin{array}{rcl} \mathbf{lunite}_{[\tau]} : 1 \bullet \tau_1 \rightarrowtail \tau & \stackrel{\text{def}}{=} & \lambda^{<}(a : 1 \bullet \tau). \\ & & \quad \mathtt{let}\, a_1 \bullet a_2 = a \\ & & \quad \mathtt{in}\, \mathtt{let}\, * = a_1 \\ & & \quad \mathtt{in}\, a_2 \end{array}$$

More interestingly, it is possible to define coercions which left or right associate fuses in memory, explicitly witnessing the underlying isomorphism.

$$\begin{array}{rcl} \mathbf{lassoc}_{[\tau_1, \tau_2, \tau_3]} & : & \tau_1 \bullet (\tau_2 \bullet \tau_3) \rightarrowtail (\tau_1 \bullet \tau_2) \bullet \tau_3 \\ & \stackrel{\text{def}}{=} & \lambda^{<}(a : \tau_1 \bullet (\tau_2 \bullet \tau_3)) \\ & & \quad \mathtt{let}\, a_1 \bullet a_{23} = a\, \mathtt{in} \\ & & \quad \mathtt{let}\, a_2 \bullet a_3 = a_{23}\, \mathtt{in} \\ & & \quad (a_1 \bullet a_2) \bullet a_3 \end{array}$$

6.4 Frontier parameters

In the core orderly lambda calculus as presented in Section 4, there is no way to define functions that are parametric over the frontier. Functions may have free references to ordered variables, or they may reserve space in their bodies, but they may not take space on the frontier as an explicit argument. This functionality can be added into the language by adding a sort of ordered lambda at the term level.

$$\begin{array}{rcl} \tau & ::= & \ldots \mid \tau_1 \stackrel{M}{\rightarrowtail} \tau_2 \mid \tau_1 \stackrel{M}{\twoheadrightarrow} \tau_2 \\ M & ::= & \ldots \mid \lambda^{<}(a{:}\tau).M \mid M \;^{<}\, Q \\ & & \mid \lambda^{>}(a{:}\tau).M \mid M \;^{>}\, Q \end{array}$$

The new arrow types classify left and right functions mapping coercions to terms. As with the ordinary arrow, the size of both of these new types is considered to be one: $|\tau_1 \stackrel{M}{\rightarrowtail} \tau_2| = |\tau_1 \stackrel{M}{\twoheadrightarrow} \tau_2| = 1$.

The typing rules are as before.

$$\frac{\Gamma; a{:}\tau_1, \Omega \vdash_{\text{trm}} M : \tau_2}{\Gamma; \Omega \vdash_{\text{trm}} \lambda^{<}(a{:}\tau_1).M : \tau_1 \stackrel{M}{\rightarrowtail} \tau_2}$$

$$\frac{\Gamma; \Omega_2 \vdash_{\text{trm}} M : \tau_1 \stackrel{M}{\rightarrowtail} \tau_2 \quad \Omega_1 \vdash_{\text{crc}} Q : \tau_1}{\Gamma; \Omega_1, \Omega_2 \vdash_{\text{trm}} M \;^{<}\, Q : \tau_2}$$

The rules for the right lambda and application are analogous.

$$\frac{\Gamma;\Omega, a{:}\tau_1 \vdash_{\mathrm{trm}} M : \tau_2}{\Gamma;\Omega \vdash_{\mathrm{trm}} \lambda^{>}(a{:}\tau_1).M : \tau_1 \overset{M}{\twoheadrightarrow} \tau_2}$$

$$\frac{\Gamma;\Omega_1 \vdash_{\mathrm{trm}} M : \tau_1 \overset{M}{\twoheadrightarrow} \tau_2 \quad \Omega_2 \vdash_{\mathrm{crc}} Q : \tau_1}{\Gamma;\Omega_1,\Omega_2 \vdash_{\mathrm{trm}} M \ {}^{>}\ Q : \tau_2}$$

Using this lambda, we can give a definition of the **pair** function which expects space already allocated on the frontier.

$$\begin{array}{l} \mathbf{pair}' : \mathtt{NS} \overset{M}{\rightarrowtail} \mathtt{NS} \overset{M}{\rightarrowtail} \tau_1 \to \tau_2 \to {!}(\tau_1 \bullet \tau_2) \overset{\mathrm{def}}{=} \\ \qquad \lambda^{<}(a_1{:}\mathtt{NS}).\lambda^{<}(a_2{:}\mathtt{NS}).\lambda(x_1 : \tau_1).\lambda(x_2 : \tau_2). \\ \qquad\qquad \mathtt{let}\, a_1 := x_1 \ \mathtt{as}\ a_1' \\ \qquad\qquad \mathtt{in}\, a_2 := x_2 \ \mathtt{as}\ a_2' \\ \qquad\qquad \mathtt{in}\, \mathtt{alloc}\, a_1' \bullet a_2' \ \mathtt{as}\ x \\ \qquad\qquad \mathtt{in}\, \mathtt{ret}\, x \end{array}$$

6.5 Future work

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.

An ad-hoc treatment of arrays can be fairly easily integrated into the orderly lambda calculus by adding all of the array operations as primitives (including the allocation and initialization operations). This is highly unsatisfactory since the intention is to make all allocation explicit through the same mechanism. A more elegant possibility is to use a dependent type formalism [25] or a type analysis formalism [5] to introduce a notion of dynamic extent into the type system. The size of an object can then be expressed as depending on the runtime value of an index object for allocation purposes, and inductive operations for traversing and initializing such objects using their indices could be defined.

Another important area for future research is to attempt to account for pointers into the frontier, or between objects in the frontier. 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 [10].

7 Related work

Ordered logic and ordered type theory have been explored extensively by Pfenning and Polakow [18, 17, 16]. Among their results include an account of various implementation properties of CPS terms [19] given in terms of a linear logical framework. However, to the best of our knowledge, no previous work has been done using ordered type theory for the purposes discussed here.

There is a significant amount of previous work applying ordinary linear type theory to memory management [23, 6, 8]. Most of this work on linear logic and memory management focuses on allowing explicit de-allocation and in-place re-use of memory, or making garbage collection more effective by propagating information about how often values can be used. Chirimar, *et al.* use linear type theory to prove the correctness of a reference counting garbage collector [2, 1].

None of this work addresses (nor is intended to address) the underlying question of separating out allocation and initialization, and of giving a foundational account of data layout. In particular, all of the above papers view allocation and initialization as an atomic operation.

The work that most closely addresses the issues that we discuss here is the alias type formalism of Smith, Walker, and Morrisett [21, 24]). 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.

References

[1] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Proving memory management invariants for a language based on linear logic. In *LISP and Functional Programming*, pages 139–150, 1992.

[2] 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.

[3] 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.

[4] 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.

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

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

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

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

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

[24] David Walker and Greg Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071, 2001.

[25] 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

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

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

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

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

Small (register) types $\boxed{\vdash \tau : \mathtt{T}_{\mathbf{reg}}}$

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

Large (heap) types $\boxed{\vdash \tau : \mathtt{T}_{\mathbf{h}}}$

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

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

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

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

$$\frac{\Gamma(x) = \tau}{\Gamma; \cdot \vdash_{\mathrm{trm}} x : \tau} \qquad \frac{}{\Gamma; \cdot \vdash_{\mathrm{trm}} \overline{n} : \mathtt{int}} \qquad \frac{}{\Gamma; \cdot \vdash_{\mathrm{trm}} \mathtt{ns} : \mathtt{NS}}$$

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

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

$$\frac{}{\vdash_{\text{val}} \overline{n} : \mathtt{int}} \qquad \frac{}{\vdash_{\text{val}} \mathtt{ns} : \mathtt{NS}} \qquad \frac{}{\vdash_{\text{val}} * : 1}$$

$$\frac{\vdash_{\text{val}} V : \tau}{\vdash_{\text{val}} !V : !\tau} \qquad \frac{\vdash \tau : \mathtt{T}_{\text{reg}} \quad x{:}\tau; \cdot \vdash_{\text{exp}} E : \tau'}{\vdash_{\text{val}} \lambda(x{:}\tau).E : \tau \to \tau'} \qquad \frac{\vdash_{\text{val}} V_1 : \tau_1 \quad \vdash_{\text{val}} V_2 : \tau_2}{\vdash_{\text{val}} V_1 \bullet V_2 : \tau_1 \bullet \tau_2}$$

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

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

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

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

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

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

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

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

B Dynamic semantics

Definitions

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

$$\begin{array}{lcl} *[\cdot] & = & * \\ a[a \mapsto V] & = & V \\ (Q_1 \bullet Q_2)[\omega_1, \omega_2] & = & Q_1[\omega_1] \bullet Q_1[\omega_2] \end{array}$$

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

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

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

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

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

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

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

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

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

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

C Proof of Theorem 1

C.1 Lemmas

It is easy to show that ordered context and frontier concatenation is associative.

Lemma 1 (Associativity)
$\Omega_1, (\Omega_2, \Omega_3) = (\Omega_1, \Omega_2), \Omega_3$ *and* $\omega_1, (\omega_2, \omega_3) = (\omega_1, \omega_2), \omega_3$

Proof: By induction on Ω_1 and ω_1. ∎

Concatenation also preserves frontier typedness.

Lemma 2 (Concatenation)
If $\vdash \omega_L : \Omega_L$ *and* $\vdash \omega_R : \Omega_R$ *and* $\vdash \Omega_L, \Omega_R$ *then* $\vdash \omega_L, \omega_R : \Omega_L, \Omega_R$

Proof: By induction on Ω_L. ∎

The converse also holds.

Lemma 3 (Splitting)
If $\vdash \omega : \Omega_L, \Omega_R$ *then* $\exists \omega_L, \omega_R$ *such that* $\omega = \omega_L, \omega_R$ *and* $\vdash \omega_L : \Omega_L$ *and* $\vdash \omega_R : \Omega_R$.

Proof: By induction on Ω_L. ∎

Several of the proof steps rely on the fact that an ordered context is uniquely split by a non-empty sub-context.

Lemma 4 (Decomposition)
If $\vdash \omega_L, \omega, \omega_R : \Omega_L, \Omega, \Omega_R$ *and* $\vdash \omega : \Omega$ $(\Omega \neq \cdot)$ *then* $\vdash \omega_L : \Omega_L$ *and* $\vdash \omega_R : \Omega_R$.

Proof: By induction on Ω_L and Ω. ∎

Lemma 5 (Frontier Substitution)
If $\vdash \omega_L, \omega, \omega_R : \Omega_L, \Omega, \Omega_R$ *and* $\Omega \vdash_{crc} Q : \tau$ *then*

1. *if* $Q[\omega] = V$ *then* $\vdash \omega : \Omega$ *and* $\vdash_{val} V : \tau$.
2. *if* $\vdash \omega : \Omega$ *then* $Q[\omega] = V$ *and* $\vdash_{val} V : \tau$.

Proof: (By induction on Q, and inversion on the derivation of $\Omega \vdash_{crc} Q : \tau$).
When $Q = *$ the proof is trivial.
When $Q = a$, we proceed by an inner induction on Ω_L and inversion on the frontier typing rules.
When $Q = Q_1 \bullet Q_2$, we use inversion and the definition of $Q[\omega]$ to show that $\exists \Omega_1, \Omega_2, \omega_1, \omega 2, V_1, V_2$ such that $\Omega_i \vdash_{crc} Q_i :$ and $Q_i[\omega_i] = V_i$. We then use associativity to appeal to induction to show that $\vdash \omega_1 : \Omega_1$ and $\vdash \omega_1 : \Omega_1$. Finally we use lemma 2 to show that $\vdash \omega_1, \omega_2 : \Omega_1, \Omega_2$. ∎

Lemma 6 (Substitution)

1. *If* $\Omega \vdash_{crc} Q : \tau$ *then*
   (a) *if* $\Gamma; \Omega_L, a{:}\tau, \Omega_R \vdash_{exp} E : \tau'$
   then $\Gamma; \Omega_L, \Omega, \Omega_R \vdash_{exp} E[Q/a] : \tau'$
   (b) *if* $\Gamma; \Omega_L, a{:}\tau, \Omega_R \vdash_{trm} M : \tau'$
   then $\Gamma; \Omega_L, \Omega, \Omega_R \vdash_{trm} M[Q/a] : \tau'$
   (c) *if* $\Omega_L, a{:}\tau, \Omega_R \vdash_{crc} Q' : \tau'$
   then $\Omega_L, \Omega, \Omega_R \vdash_{crc} Q'[Q/a] : \tau'$
2. *If* $\Gamma; \cdot \vdash_{trm} M : \tau$ *then*
   (a) *if* $\Gamma_1, x{:}\tau, \Gamma_2; \Omega \vdash_{exp} E : \tau'$
   then $\Gamma_1, \Gamma_2; \Omega \vdash_{exp} E[M/x] : \tau'$
   (b) *if* $\Gamma_1, x{:}\tau, \Gamma_2; \Omega \vdash_{trm} M' : \tau'$
   then $\Gamma_1, \Gamma_2; \Omega \vdash_{trm} M'[M/x] : \tau'$

Proof: (By induction on the typing derivations of the *inner* term) ∎

Lemma 7 (Inclusion)

1. *If* $\vdash_{val} V : \tau$ *and* $\vdash \tau : \mathtt{T}_{reg}$ *then* $\cdot; \cdot \vdash_{trm} V : \tau$.
2. *If* $\cdot; \cdot \vdash_{trm} M : \tau$ *then* $\vdash_{val} M : \tau$.

Lemma 8 (Canonical Forms)

1. *if* $\cdot; \Omega \vdash_{trm} M : \tau$ *then*
   (a) *if* $\tau = \mathtt{int}$ *then* $M = \overline{n}$

(b) *if* $\tau = \texttt{NS}$ *then* $M = \texttt{ns}$

(c) *if* $\tau = \tau_1 \to \tau_2$ *then* $M = \lambda(x{:}\tau_1).E$

(d) *if* $\tau = {!}\,\tau'$ *then* $M = {!}\,V$

2. *if* $\vdash_{\scriptscriptstyle val} V : \tau_1 \bullet \tau_2$ *then* $V = V_1 \bullet V_2$

3. *if* $\Omega \vdash_{\scriptscriptstyle crc} Q : \tau$ *then*

   (a) *if* $\tau = 1$ *then either* $Q = *$ *or* $Q = a$.

   (b) *if* $\tau = \tau_1 \bullet \tau_2$ *then either* $Q = Q_1 \bullet Q_2$ *or* $Q = a$.

   (c) *if* $\tau = \texttt{int}, \tau_1 \to \tau_2, {!}\,\tau,$ *or* $\texttt{NS}$ *then* $Q = a$.

C.2 Progress & Preservation

If $\vdash \omega : \Omega$ and $\cdot;\Omega \vdash_{\text{exp}} E : \tau$ then

1. Either $(\omega, E) \mapsto (\omega', E')$ or E is a value.

2. if $(\omega, E) \mapsto (\omega', E')$ then $\exists \Omega'$ such that $\vdash \omega' : \Omega'$ and $\cdot;\Omega' \vdash_{\text{exp}} E' : \tau$

Proof: (By induction on the derivation of $\cdot;\Omega \vdash_{\text{exp}} E : \tau$)
We proceed by cases on the last rule of the derivation.

$\to$ if $\dfrac{\Gamma;\cdot \vdash_{\text{trm}} M : \tau}{\Gamma;\cdot \vdash_{\text{exp}} \texttt{ret}\, M : \tau}$ then E is a value and is well-typed.

$\to$ if $\dfrac{\cdot;\Omega \vdash_{\text{trm}} M_1 : \tau_1 \to \tau_2 \quad \cdot;\cdot \vdash_{\text{trm}} M_2 : \tau_1}{\cdot;\Omega \vdash_{\text{exp}} M_1\, M_2 : \tau_2}$ then by canonical forms (lemma 8) $M_1 = \lambda(x{:}\tau).E'$.
Therefore,

1. $(\omega, (\lambda(x{:}\tau).E')\, M_2) \mapsto (\omega, E'[M_2/x])$
2. By inversion, $x{:}\tau_1;\Omega \vdash_{\text{exp}} E' : \tau_2$ and $\cdot;\cdot \vdash_{\text{trm}} M_2 : \tau_1$ so by lemma 6 $\cdot;\Omega \vdash_{\text{exp}} E'[M_2/x] : \tau_2$

$\to$ if $\dfrac{\cdot;\Omega \vdash_{\text{exp}} E_1 : \tau_1 \quad \cdot, x{:}\tau_1;\cdot \vdash_{\text{exp}} E_2 : \tau_2}{\cdot;\Omega \vdash_{\text{exp}} \texttt{let}\, x{:}\tau_1 = E_1\, \texttt{in}\, E_2 : \tau_2}$ then by induction one of the following holds.

- $E_1 = \texttt{ret}\, M$.
  1. Then by inversion $\omega = \cdot$ and hence $(\cdot, \texttt{let}\, x{:}\tau = \texttt{ret}\, M_v\, \texttt{in}\, E_2) \mapsto (\cdot, E_2[M_v/x])$
  2. By inversion, $x{:}\tau_1;\Omega \vdash_{\text{exp}} E_2 : \tau_2$ and $\cdot;\cdot \vdash_{\text{trm}} M_2 : \tau_1$ so by lemma 6 $\cdot;\Omega \vdash_{\text{exp}} E'[M_2/x] : \tau_2$
- $\exists E_1', \omega'$ such that $(\omega, E) \mapsto (\omega', E_1')$
  1. Then $(\omega, \texttt{let}\, x{:}\tau = E_1\, \texttt{in}\, E_2) \mapsto (\omega', \texttt{let}\, x{:}\tau = E_1'\, \texttt{in}\, E_2)$
  2. By induction, $\exists \Omega'$ such that $\vdash \omega' : \Omega'$ and $\cdot;\Omega' \vdash_{\text{exp}} E_1' : \tau_1$. Therefore $\cdot;\Omega' \vdash_{\text{exp}} \texttt{let}\, x{:}\tau_1 = E_1'\, \texttt{in}\, E_2 : \tau_2$ by construction.

$\to$ if $\dfrac{\cdot; a{:}\texttt{NS}^n \vdash_{\text{exp}} E : \tau}{\cdot;\Omega \vdash_{\text{exp}} \texttt{reserve}_n\, \texttt{as} a\, \texttt{in}\, E : \tau}$ then

1. $(\omega, \texttt{reserve}_n\, \texttt{as} a\, \texttt{in}\, E) \mapsto (a \mapsto \texttt{ns}^n, E)$
2. $\vdash a \mapsto \texttt{ns}^n : a{:}\texttt{NS}^n$ and $\cdot; a{:}\texttt{NS}^n \vdash_{\text{exp}} E : \tau$.

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

1. By assumption $\vdash \omega : \Omega_L, \Omega_R$ so by lemma 3, ω can be split into ω_L, ω_R such that $\vdash \omega_L : \Omega_L$ and $\vdash \omega_R : \Omega_R$.
   By inversion, we have $\Omega_L \vdash_{\text{crc}} Q : \tau$
   Therefore, by lemma 5 $Q_v[\omega_L] = V$ and $\vdash_{\text{val}} V : \tau$. $((\omega_L, \omega_R), \texttt{alloc}\, Q\, \texttt{as}\, x\, \texttt{in}\, E \mapsto (\omega_R, E[!\,V/x])$
2. Suppose $((\omega_L, \omega_R), \texttt{alloc}\, Q\, \texttt{as}\, x\, \texttt{in}\, E \mapsto (\omega_R, E[!\,V/x])$.
   It suffices to show that $\vdash \omega_R : \Omega_R$ and $\cdot; \Omega_R \vdash_{\text{exp}} E[!\,V/x]) : \tau'$
   By inversion on the reduction, $Q_v[\omega_L] = V$
   By assumption, $\vdash \omega_L, \omega_R : \Omega_L, \Omega_R$
   Therefore, by lemma 5 $\vdash \omega_L : \Omega_L$ and $\vdash_{\text{val}} V : \tau$.
   By inversion $\cdot, x{:}\,!\,\tau; \Omega_R \vdash_{\text{exp}} E : \tau'$, so by lemma 6 $\cdot; \Omega_R \vdash_{\text{exp}} E[!\,V/x]) : \tau'$.
   If $\Omega_L \neq \cdot$ then by lemma 4 $\vdash \omega_L : \Omega_L$ implies that $\vdash \omega_R : \Omega_R$.
   If $\Omega_L = \cdot$ then $(\Omega_L, \Omega_R) = \Omega_R$ and $\vdash \omega_R : \Omega_R$ by assumption.

$\to$ if $\dfrac{\begin{array}{c}\Omega \vdash_{\text{crc}} Q : \tau \quad \vdash \tau : \mathrm{T}_{\text{reg}} \\ \cdot; \cdot \vdash_{\text{trm}} M : \tau' \quad \cdot; \Omega_L, a{:}\tau', \Omega_R \vdash_{\text{exp}} E : \tau''\end{array}}{\cdot; \Omega_L, \Omega, \Omega_R \vdash_{\text{exp}} Q := M\, \texttt{as}\, a\, \texttt{in}\, E : \tau''}$ then by lemma 8(canonical forms), $Q = a'$ and by inversion $\Omega = a'{:}\tau$.

1. By two applications of lemma 3, $\omega = \omega_L, \omega', \omega_R$ and $\vdash \omega' : a'{:}\tau$. By inversion, $\omega' = a \mapsto V$, and hence $((\omega_1, a \mapsto V, \omega_2), a := M\, \texttt{as}\, a'\, \texttt{in}\, E) \mapsto ((\omega_1, a' \mapsto M, \omega_2), E)$
2. Suppose $((\omega_1, a \mapsto V, \omega_2), a := M\, \texttt{as}\, a'\, \texttt{in}\, E) \mapsto ((\omega_1, a' \mapsto M, \omega_2), E)$
   Since $\cdot; \Omega_L, a{:}\tau', \Omega_R \vdash_{\text{exp}} E{:}\tau''$ by assumption, it suffices to show that $\vdash (\omega_1, a \mapsto M, \omega_2){:}\Omega_L, a{:}\tau', \Omega_R$.
   By assumption, $\vdash (\omega_1, a \mapsto V, \omega_2) : \Omega_L, a{:}\tau, \Omega_R$.
   By definition $a[a \mapsto V] = V$.
   By lemma 4 $\vdash \omega_l : \Omega_L$ and $\vdash \omega_R : \Omega_R$.
   By assumption $\cdot; \cdot \vdash_{\text{trm}} M : \tau'$ and by lemma 7 $\vdash_{\text{val}} \cdot; \cdot : M\tau'$
   Therefore, by construction, $\vdash a \mapsto M : a{:}\tau$.
   Finally, by two applications of lemma 2, $\vdash (\omega_1, a \mapsto M, \omega_2) : \Omega_L, a{:}\tau', \Omega_R$.

$\to$ if $\dfrac{\Omega \vdash_{\text{crc}} Q : 1 \quad \cdot; \Omega_L, \Omega_R \vdash_{\text{exp}} E : \tau}{\cdot; \Omega_L, \Omega, \Omega_R \vdash_{\text{exp}} \texttt{let} * = Q\, \texttt{in}\, E : \tau}$ then by canonical forms either $Q = *$ or $Q = a$.

- if $Q = *$ then $(\omega, \texttt{let} * = *\, \texttt{in}\, E) \mapsto (\omega, E)$
- if $Q = a$ then since $((\omega_L, a \mapsto *, \omega_R), \texttt{let} * = a\, \texttt{in}\, E) \mapsto ((\omega_L, \omega_R), E)$ it suffices to show that $\omega = (\omega_L, a \mapsto *, \omega_R)$ for some ω_L, ω_R. This follows by two applications of lemma 3 and inversion.

$\to$ if $\dfrac{\begin{array}{c}\Omega \vdash_{\text{crc}} Q : \tau \\ \cdot; \Omega_1, a{:}\tau, \Omega_2 \vdash_{\text{exp}} E : \tau'\end{array}}{\cdot; \Omega_1, \Omega, \Omega_2 \vdash_{\text{exp}} \texttt{let}\, a = Q\, \texttt{in}\, E : \tau'}$ then $(\omega, \texttt{let}\, a = Q_v\, \texttt{in}\, E) \mapsto (\omega, E[Q_v/a])$

$\to$ if $\dfrac{\begin{array}{c}\Omega \vdash_{\text{crc}} Q : \tau_1 \bullet \tau_2 \\ \cdot; \Omega_1, a_1{:}\tau_1, a_2{:}\tau_2, \Omega_2 \vdash_{\text{exp}} E : \tau\end{array}}{\cdot; \Omega_1, \Omega, \Omega_2 \vdash_{\text{exp}} \texttt{let}\, a_1 \bullet a_2 = Q\, \texttt{in}\, E : \tau}$ then by canonical forms, $Q = Q_1 \bullet Q_2$ or $Q = a$.

- if $Q = Q_1 \bullet Q_2$ then $(\omega, \texttt{let}\, a_1 \bullet a_2 = Q_1 \bullet Q_2\, \texttt{in}\, E) \mapsto (\omega, E[Q_1, Q_2/a_1, a_2])$

– if $Q = a$ then by inversion $\Omega = a{:}\tau_1 \bullet \tau_2$. By lemma 3, $\omega = \omega_1, \omega', \omega_2$ and $\vdash \omega' : \Omega$. By inversion, $\omega' = a \mapsto V$ and $\vdash_{\text{val}} V : \tau_1 \bullet \tau_2$. By canonical forms, $V = V_1 \bullet V_2$. Therefore, $((\omega_1, a \mapsto V_1 \bullet V_2, \omega_2), \texttt{let}\, a_1 \bullet a_2 = a\, \texttt{in}\, E) \mapsto ((\omega_1, a_1 \mapsto V_1, a_2 \mapsto V_2, \omega_2), E)$

→ if $\dfrac{\cdot;\cdot \vdash_{\text{trm}} M : !(\tau_1 \bullet \tau_2) \quad \cdot, x_1{:}!\tau_1, x_2{:}!\tau_2; \Omega \vdash_{\text{exp}} E : \tau}{\cdot;\Omega \vdash_{\text{exp}} \texttt{let!}\, x_1 \bullet x_2 = M\, \texttt{in}\, E : \tau}$ then by canonical forms, $M = !V$ and $V = V_1 \bullet V_2$. Hence $(\omega, \texttt{let!}\, x_1 \bullet x_2 = !(V_1 \bullet V_2)\, \texttt{in}\, E) \mapsto (\omega, E[!V_1, !V_2/x_1, x_2])$

→ if $\dfrac{\cdot;\cdot \vdash_{\text{trm}} M : !\tau_1 \quad \vdash \tau_1 : \mathtt{T}_{\text{reg}} \quad \cdot, x{:}\tau_1, ; \Omega \vdash_{\text{exp}} E : \tau_2}{\cdot;\Omega \vdash_{\text{exp}} \texttt{let}\, !x = M\, \texttt{in}\, E : \tau_2}$ then by canonical forms, $M = !V$ and hence $(\omega, \texttt{let}\, !x = !V\, \texttt{in}\, E) \mapsto (\omega, E[V/x])$

■