Linear Types for Packet Processing

Robert Ennals[1], Richard Sharp[2], and Alan Mycroft[1]

[1] Computer Laboratory, Cambridge University
15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK.
{robert.ennals,am}@cl.cam.ac.uk
[2] Intel Research Cambridge,
15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK.
richard.sharp@intel.com

Abstract. We present PacLang: an imperative, concurrent, linearly-typed language designed for expressing packet processing applications. PacLang's linear type system ensures that no packet is referenced by more than one thread, but allows multiple references to a packet *within a thread*. We argue (*i*) that this property greatly simplifies compilation of high-level programs to the distributed memory architectures of modern Network Processors; and (*ii*) that PacLang's type system captures that style in which imperative packet processing programs are already written. Claim (*ii*) is justified by means of a case-study: we describe a PacLang implementation of the IPv4 unicast packet forwarding algorithm.
PacLang is formalised by means of an operational semantics and a Unique Ownership theorem formalises its correctness with respect to the type system.

1 Introduction

Network Processors (NPs) [1,10,17] are programmable, application-specific hardware architectures designed to facilitate high-speed packet processing. NPs typically contain multiple processor cores (allowing multiple network packets to be processed concurrently) and multiple memory banks. The majority of NPs are based on what is known as a *distributed memory architecture*: not all memory banks are accessible from all processor cores.

State-of-the-art programming techniques for NPs are invariably low-level, requiring programmers to specify explicitly how data will be mapped to, and dynamically moved between, complex arrays of available memory banks. This results in design flows which are difficult to master, error prone and inextricably tied to one particular architecture. Our research attempts to address this problem by developing higher-level languages and tools for programming NPs.

This paper formally defines an imperative language which provides an architecturally neutral concurrent programming model—multiple parallel threads accessing a shared heap—and considers the problem of compiling this language to NP-based architectures. We argue that although, *in general*, the efficient compilation of a high-level concurrent language to a diverse range of NP architectures is very difficult, *in the domain of packet processing* the problem is more

D.A. Schmidt (Ed.): ESOP 2004, LNCS 2986, pp. 204–218, 2004.
© Springer-Verlag Berlin Heidelberg 2004

tractable. Our argument is based on the observation that a great deal of network processing code is written in a restricted data-flow style, in which network packets can be seen to "flow" through a packet processing system. We formalise this restricted style of programming by means of a linear type system and state a *unique ownership property*: if a program is typeable then at any given point in its execution, each packet in the heap is referenced by exactly one thread. It is this property that we argue simplifies the process of compiling for network processors.

Contributions of this paper: (*i*) the formalisation of a concurrent, first-order, imperative language which we argue is well suited for compiling to a diverse range of NP architectures; (*ii*) a simple, intuitive linear type system in which no packet is referenced by more than one thread but multiple references to packets *within a thread* are allowed; and (*iii*) a case-study that demonstrates both that the type system is applicable to the domain of packet processing (i.e. not overly restrictive) and that the resulting code is comprehensible to C/C++ programmers.

Previous research [16] has shown that (*i*) locking is unnecessary for linearly typed objects; and (*ii*) memory leaks can be detected statically as type errors. Although these are not direct contributions of this paper, they are nevertheless benefits enjoyed by PacLang programmers.

Structure of this paper: We outline the design of an NP-based platform and explain how our unique ownership property helps us target such architectures (Section 1.1). Section 2 gives formal syntax and types for PacLang and Section 3 defines its linear typing judgements along with more intuitive explanation. An operational semantics for the language is presented and the *unique ownership property* formalised (Section 4). To justify that our linear type system is not overly restrictive in the domain of packet processing applications, we present a case study which shows how PacLang can be used to implement an IPv4 packet forwarder (Section 5). Finally we consider related work (Section 6) and conclude, giving directions for future research/development (Section 7).

1.1 Compiling for Network Processors

Figure 1 gives a concrete example of an NP-based system: the Intel IXDP2400 [9]. The platform consists of two Intel IXP2400 NPs [10] connected in a pipeline configuration. Each NP contains 16 small RISC processors called *micro-engines* (labelled μE-1,. . .,μE-16), a single Intel X-Scale processor and some on-chip shared RAM. Each micro-engine[1] has its own local memory. The external SRAM and DRAMs can be accessed by all the cores on the NP to which they are connected. Micro-engines are equipped with *next neighbour registers*. The next neighbour

[1] Note that each micro-engine actually supports 8 hardware-level threads. This level of detail is beyond the scope of the paper.

registers of μE-i are accessible to μE-$(i+1)$, forming a uni-directional datapath between consecutive micro-engines.

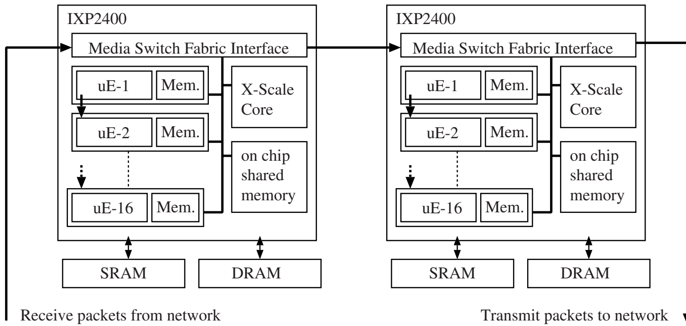

Fig. 1. Simplified diagram of IXDP2400 NP evaluation board

When programming for architectures such as this a great deal of low-level knowledge is required to decide how packets should be represented in memory and how packets should be transferred between different memory banks[2]. For example, if a small amount of data needs to be transferred between two threads running on μE-1 and μE-2, then the most efficient mechanism may well be to move the data across the serial datapath connecting μE-1 and μE-2. Conversely, if we have data already in DRAM that needs to be transferred between threads running on μE-1 and the X-Scale core then leaving the data where it is and passing a reference to it is the obvious choice. When passing data between the two separate IXP2400s we *must* pass-by-value since the two processors have no memories in common.

Clearly, if we are to succeed in our goal of designing a high-level language that can be compiled to a variety of different NP architectures then a compiler must be *free to choose* how data will be passed between threads (e.g. pass a reference pointing to shared memory, or pass the data itself over a shared bus). Unfortunately, conventional imperative languages (e.g. C/C++) are unsatisfactory in this respect since they force the programmer to *explicitly specify* whether data will be passed by value or passed by reference. Note that a compiler cannot simply transform pass-by-value into pass-by-reference (or vice versa); sophisticated *alias analysis* is required to determine whether this transformation is semantics-preserving—an analysis that is undecidable in general.

In contrast, in the PacLang framework, the unique ownership property (provided by our linear type system) gives us two key benefits: (i) a thread

[2] Moving a packet between different memories as it "flows" through a packet processing application is crucial to achieving efficient implementation on the IXP (and other NPs). Data should be located "as close as possible" to the thread operating on it in order to reduce memory latency.

$v \leftarrow i$		*integer constant*
$\quad \mid\ b$		*boolean constant*
$\quad \mid\ x$		*variable*
$e \leftarrow$ `if` v `then` e_1 `else` e_2		*conditional*
$\quad \mid$ `let` $(x_1, \ldots, x_j) = f(v_1, \ldots, v_k)$ `in` e	$(j, k \geq 0)$	*call*
$\quad \mid$ `return` $(v_1, \ldots, v_k)$	$(k \geq 0)$	*return*
$d \leftarrow (\sigma_1, \ldots, \sigma_j)\ f(\rho_1\ x_1, \ldots, \rho_k\ x_k)\{e\}$	$(j, k \geq 0)$	*function definition*
$\quad \mid$ *packetQueue* q;		*queue definition*
$p \leftarrow d_1 \ldots d_n$		*top-level program*
$\rho,\ \sigma,\ \tau \leftarrow$ *int* | *bool* | *packet* | *!packet*		*types*

Fig. 2. Syntax of PacLang, our simple first-order CBV language.

can dynamically move an object it owns to a new location without worrying about dangling references; and (*ii*) pass-by-value and pass-by-reference are semantically indistinguishable when transferring data between threads. This gives a PacLang compiler the freedom to make decisions regarding both the flow of packets between available memory banks and the low-level mechanisms employed to implement these transfers, on a per-architecture basis.

2 A Packet Processing Language

PacLang is a concurrent, first-order, imperative language with a linear type system; its syntax is presented in Figure 2. Concurrency is supported by the simple expedient of treating functions whose name is spelt $\texttt{main}_1, \ldots, \texttt{main}_n$, as representing the entry points of n static threads $(n \geq 1)$[3]. Such functions take no arguments and return no results. Statically allocated, synchronous (blocking) packet queues are provided to facilitate inter-thread communication.

Expressions, e, comprise conditionals, function calls and `return`s. Note that PacLang syntax forces all function return values to be explicitly bound to variables (cf. A-Normal Form [7]); this simplifies subsequent typing and semantic rules. In addition to function names defined by *function definitions* we assume some functions to be built-in, e.g. arithmetic and queue manipulation primitives (see later); both forms may be called using the `let` construct. We use v to range over simple expressions (constants and variables). A program, p, consists of a sequence of function and queue definitions, d. Note that functions take multiple arguments and return multiple results.

In contrast to most languages in the linear types literature, PacLang has imperative features. Given a packet reference p, an integer offset i and a value

[3] In practice a compiler may wish to transform a single programmer-specified thread into several finer-grained threads for efficient implementation on certain NP architectures. Such *automated partitioning* is beyond the scope of this paper.

v, the built-in function `update(`p`,`i`,`v`)` updates p to contain v at offset i. The packet read function `read(`p`,`i`)` returns the value stored at offset i in packet p.

Global queues also provide side-effecting enqueue and dequeue operations. To simplify the presentation we assume built-in `enq` and `deq` functions replicated over queue names. We write `q.enq(`p`)` to enqueue a packet p in the queue declared with name `q`. Similarly, `q.deq()` returns a packet dequeued from `q`. The semantics of queues are formalised in Section 4.

We also provide built-in functions `new(`i`)` which given an integer size i, creates a new packet buffer of size i, returning its associated *packet reference*; and `kill(`p`)`, which takes a packet reference p, consumes it, deallocates the associated packet and returns nothing. In PacLang packets are simply dynamically created arrays. As well as representing network packets, they can also be used as containers for arbitrary (non-pointer) data within a PacLang program.

Syntactic sugaring is employed to increase readability for those familiar with C. We write "$f(v_1, \ldots, v_k); e$" to abbreviate "`let` $() = f(v_1, \ldots, v_k)$ `in` e". We allow nested function calls to be written directly, assuming a left-to-right evaluation order. For example, the expression "`let x =` $f(g(x), h(y))$ `in` e", becomes "`let` z_1 `=` $g(x)$ `in let` z_2 `=` $h(y)$ `in let x =` $f(z_1, z_2)$ `in` e" where $z_1, z_2 \notin fv(e)$. We add types to the variables following `let` (corresponding to the return type of the called function) as usual in C, and also drop the `let` keyword and parentheses entirely. We write `void` in place of `()` in the signature of functions that return an empty tuple of values and adopt the convention that an expression without an explicit `return` implicitly returns a value of type `void`. We also introduce a `switch`/`case` construct which is translated into nested `if`s in the usual way. We write `update(`p`,`a`,`v`)` using C-like syntax, p`[`a`] =` v, and use array subscript notation, p`[`a`]`, for `read(`p`,`a`)`. To avoid the extra syntax required for explicit while-loops, we use recursion to model iteration in this presentation. (Of course, a PacLang compiler would implement tail-call optimisation to ensure the generated code is as efficient as if it had been written with while-loops.)

3 Linear Type System

An industrial NP programming textbook [11] observes: *"In most designs processing of a packet by multiple threads simultaneously is rare, but using multiple threads to process packets at different points in the packet's lifetime is common."* This quote summarises the philosophy behind PacLang's linear type system: we (i) *guarantee* that, in a well typed program, a packet, p, cannot be accessed by multiple threads simultaneously; whilst (ii) allowing p to be transferred between threads as it flows through an application. Unlike many linear type systems, we allow p to be referenced multiple times (aliasing). However, we require that all references to p reside in the same thread. Allowing intra-thread aliasing is crucial to the readability of PacLang code. It is this property that allows PacLang programmers to write in the familiar imperative (pass-by-reference) style—see

Section 5. In many ways, PacLang's type system is inspired by Kobayashi's work on *Quasi-Linear Types* [12].

We start by defining some terminology. A variable, x, that holds a reference to a packet p, will either be of type *packet* or *!packet*:

- If x has type *packet* then, at any point where x is in scope, there are no other references to p. We say that x is an *owning variable* and holds an *owning reference* for p. If one has an owning reference to p then one can safely move p to another thread without creating dangling references.
- If x has type *!packet* then other references to p may exist, but they must be on the same thread as x. We say that x is an *alias* for p. An alias cannot escape from its enclosing scope.

Value types, *int* and *bool*, are non-linear and behave as is normal.

Our type rules use arrow-notation for vectors. We write $\vec{\tau}$ to represent a vector of types and similarly $\vec{x}$ and $\vec{v}$ for vectors of variables and values respectively. We let ρ, σ, τ range over types. Function types are written $\rho_1 \times \ldots \times \rho_k \rightarrow \sigma_1 \times \ldots \times \sigma_j$ or more briefly as $\vec{\rho} \rightarrow \vec{\sigma}$. Given a function name f, $\mathcal{F}(f)$ gives its type signature. For user-defined functions this comes from the textual definition; the types of built-in functions are given in Figure 4.

Typing judgements for expressions ($\overset{e}{\vdash}$) take the form $\Gamma \overset{e}{\vdash} e : \vec{\tau}$ where Γ is a typing environment, e is an expression (generally returning a tuple) and $\vec{\tau}$ is the return type (generally a tuple type). Γ can be regarded as the set of *capabilities* that e requires during its execution. Each capability in Γ takes the form $x : \rho$ specifying that x is used as ρ in e. We adopt the usual notation of writing $\Gamma, x{:}\tau$ to represent Γ extended to include the capability $x{:}\tau$ (note that scope shadowing is problematic in linear type systems so we will simply presume that all variable names are distinct).

The typing rules for PacLang, rely on two partial binary operators, "+" and ";" which operate on type environments. An expression which requires capabilities in Γ_1 and then subsequently the capabilities in Γ_2, can equivalently be said to require the capabilities in environment $\Gamma_1; \Gamma_2$. Similarly, if an expression requires capabilities in both Γ_1 and Γ_2 in an unordered fashion, it requires $\Gamma_1 + \Gamma_2$. We first define, "+" and ";" over *types*:

$$\begin{array}{cc} \mathit{int} + \mathit{int} = \mathit{int} & \mathit{int} \;;\; \mathit{int} = \mathit{int} \\ \mathit{bool} + \mathit{bool} = \mathit{bool} & \mathit{bool} \;;\; \mathit{bool} = \mathit{bool} \\ \mathit{!packet} + \mathit{!packet} = \mathit{!packet} & \mathit{!packet} \;;\; \mathit{!packet} = \mathit{!packet} \\ & \mathit{!packet} \;;\; \mathit{packet} = \mathit{packet} \end{array}$$

If no rule is given then that signifies a type error. Note that ";" is a strict superset of "+". We extend these operators to type environments by the following definitions (here $\otimes$ ranges over "+" and ";"):

$$\begin{aligned} \Gamma_1 \otimes \Gamma_2 \overset{def}{=} & \{x : \tau \mid x : \tau \in \Gamma_1 \wedge x \notin \text{dom } \Gamma_2\} \quad \cup \\ & \{x : \tau \mid x : \tau \in \Gamma_2 \wedge x \notin \text{dom } \Gamma_1\} \quad \cup \\ & \{x : \tau_1 \otimes \tau_2 \mid x : \tau_1 \in \Gamma_1 \wedge x : \tau_2 \in \Gamma_2\} \end{aligned}$$

$$\frac{}{\{x:\tau\} \overset{v}{\vdash} x:\tau}\,(var) \qquad \frac{}{\emptyset \overset{v}{\vdash} i: int}\,(int)\ (i \in \mathbb{Z}) \qquad \frac{}{\emptyset \overset{v}{\vdash} b: bool}\,(bool)\ (i \in \mathbb{B})$$

$$\frac{\Gamma_1 \overset{v}{\vdash} v_1:\rho_1 \quad \ldots \quad \Gamma_k \overset{v}{\vdash} v_k:\rho_k \qquad \Gamma_0, x_1{:}\sigma_1, \ldots, x_j{:}\sigma_j \overset{e}{\vdash} e:\vec{\tau}}{(\Gamma_1 + \ldots + \Gamma_k); \Gamma_0 \overset{e}{\vdash} \texttt{let}\ (x_1, \ldots, x_j) = f(v_1, \ldots, v_k)\ \texttt{in}\ e:\vec{\tau}}\,(apseq)$$

provided $\mathcal{F}(f) = (\rho_1 \times \ldots \times \rho_k) \to (\sigma_1 \times \ldots \times \sigma_j)$

$$\frac{\Gamma_1 \overset{v}{\vdash} v_1:\tau_1 \quad \ldots \quad \Gamma_j \overset{v}{\vdash} v_j:\tau_j}{\Gamma_1 + \ldots + \Gamma_j \overset{e}{\vdash} \texttt{return}\ (v_1, \ldots, v_j): \tau_1 \times \ldots \times \tau_j}\,(ret)$$

provided no τ_j is *!packet*

$$\frac{\Gamma_1 \overset{v}{\vdash} v: bool \quad \Gamma_2 \overset{e}{\vdash} e_1:\vec{\tau} \quad \Gamma_2 \overset{e}{\vdash} e_2:\vec{\tau}}{\Gamma_1;\Gamma_2 \overset{e}{\vdash} \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2:\vec{\tau}}\,(if) \qquad \frac{\Gamma \overset{e}{\vdash} e:\vec{\tau}}{\Gamma, x:\rho \overset{e}{\vdash} e:\vec{\tau}}\,(weak)$$

provided ρ is not *packet*

$$\frac{\{x_1{:}\rho_1, \ldots, x_k{:}\rho_k\} \overset{e}{\vdash} e:\sigma_1 \times \ldots \times \sigma_j}{\overset{d}{\vdash} (\sigma_1, \ldots, \sigma_j)\ f(\rho_1\ x_1, \ldots, \rho_k\ x_k)\{e\}}\,(fundef) \qquad \frac{}{\overset{d}{\vdash} packetQueue\ q;}\,(qdef)$$

$$\frac{\overset{d}{\vdash} d_1 \quad \ldots \quad \overset{d}{\vdash} d_n}{\overset{p}{\vdash} d_1 \ldots d_n}\,(prog)$$

Fig. 3. Linear typing rules

Figure 3 presents the linear type rules. Typing rules for definitions ($\overset{d}{\vdash}$), programs ($\overset{p}{\vdash}$) and simple expressions ($\overset{v}{\vdash}$) are self-explanatory, although note that, in contrast to non-linear type systems, the (*var*) rule only holds if the environment contains the *single* assumption, $x{:}\tau$.

The typing rules for expressions ($\overset{e}{\vdash}$) deserve more detailed explanation. Essentially they ensure (*i*) that an owning variable x for a packet p goes out of scope for the lifetime of any aliases to p; and (*ii*) that an owning variable x may no longer be used if its owning reference is passed to another owning variable.

For a variable x to be of type *!packet*, x must be bound as a formal parameter of its enclosing function definition. When calling a function, f, the (*apseq*) rule allows aliases to be created from an owning variable provided that the owning variable goes out of scope[4] during the call, and that the alias dies before the function returns. It is the use of the environment sequencing operator ";" in the consequent of the (*apseq*) rule that facilitates the creation of aliases from an owning variable. (Recall that *!packet* ; *packet* = *packet*).

The (*ret*) rule types its subexpressions in environments $\Gamma_1, \ldots, \Gamma_j$ respectively, and uses the "+" operator to combine these environments in the antecedent. The "+" operator is used in this context to prevent owning references

[4] In PacLang this is ensured by the absence of nested function definitions and the absence of global packet variables.

being duplicated. We prevent values of type *!packet* being returned because the enclosing scope may contain the corresponding owning variable.

```
int ⟨arith-op⟩ (int x, int y);        bool ⟨bool-op⟩ (bool x, bool y);
bool ⟨rel-op⟩ (int x, int y);
packet new(int size);                  void kill(packet p);
packet q.deq();                        void q.enq(packet p);
int read(!packet p, int offset);
void update(!packet p, int offset, int value);
```

Fig. 4. Type signatures of the built-in functions expressed in C-like syntax

The type signatures of built-in functions are given in Figure 4. Note that inter-thread communication primitives, *q.enq* and *q.deq*, enqueue and dequeue values of type *packet* (i.e. owning references). Thus when transferring a packet between threads, the packet can be physically relocated.

We assume the existence of special queues, `receive` and `transmit` which represent the external network interface. `receive.deq()` reads a packet from the network; `transmit.enq(p)` schedules packet `p` for transmission. In reality a network appliance would typically contain multiple network interfaces and one would expect a variety of more sophisticated range of read/write calls. However, since this does not affect the fundamental concepts expressed in PacLang, we stick to the simple interface above for the purposes of this paper.

The following two examples (in de-sugared form) help to illustrate PacLang's linear type system. First consider the following valid PacLang expression:

```
let packet x = new(10) in
let () = update(x,0,5) in
let () = update(x,1,10) in
let () = transmit.enq(x) in return ()
```

This code will type check successfully as, although `x` is accessed multiple times, `update` only requires a local alias to `x` (*!packet*) and so does not prevent `x` being passed as type *packet* to `transmit.enq`.

Conversely, consider the following *invalid* PacLang expression:

```
let packet x = new(10) in
let () = kill(x) in
let () = transmit.enq(x) in return ()
```

The type system rejects this program since `x` is passed twice as type *packet*: once to the `kill` function and then again to the `transmit.enq` function. Once `x` has been passed as type *packet*, it effectively disappears and cannot be used again.

4 Operational Semantics

In this section we define the language formally by means of a small-step operational semantics modelled on the Chemical Abstract Machine [2]. States of the machine are a chemical solution of reacting molecules. We let W range over machine states. The "|" operator separates molecules in the multiset solution (cf. parallel composition). Molecules take the following forms:

$$
\begin{array}{rll}
M \leftarrow & [\![\{\vec{x}\}e]\!]_f & \text{(function definition)} \\
\mid & \langle\!\langle Q \rangle\!\rangle_q & \text{(queue)} \\
\mid & \langle H \rangle & \text{(shared heap)} \\
\mid & (\!| e, \Sigma |\!) & \text{(thread: expression and its stack frames)}
\end{array}
$$

We let α range over an infinite set of heap-bound packet reference values[5]. We extend the syntactic category, v, allowing values also to be packet references, α, in addition to integer and boolean constants.

A *function molecule*, $[\![\{\vec{x}\}e]\!]_f$, corresponds to a source-level function definition for f, with formal parameters, $\vec{x}$, and body e. Function molecules themselves do not perform computation, they exist merely to provide their bodies to callers.

A *queue molecule*, $\langle\!\langle Q \rangle\!\rangle_q$ has name q and contents Q, where Q takes the form of a list of elements separated by the associative operator $\bullet$. An empty queue is written ϵ. In the semantics presented here, queues are unbounded with synchronous (blocking) behaviour.

A *heap molecule*, $\langle H \rangle$, maps *packet references*, α, onto packets, p. A packet, p, is a finite mapping from integer offsets, i, to integer values, $p(i)$[6]. We define $empty_k$ as the empty packet of size k: $empty_k = \{0 \mapsto 0, \ldots, (k-1) \mapsto 0\}$. We write $\langle H \backslash \alpha \rangle$ for the heap which is as $\langle H \rangle$ but no longer contains an entry for α. The heap, which is as $\langle H \rangle$ but maps α to p, is written $\langle H[\alpha \mapsto p] \rangle$. Thus, updating the ith element of the packet referenced by α to value 42 is written $\langle H[\alpha \mapsto H(\alpha)[i \mapsto 42]] \rangle$. The reader should note that the heap molecule need not be implemented as a single shared memory. The central result of this paper is that, due to PacLang's unique ownership property, a compiler is free to map the packets in $\langle H \rangle$ to the multiple memory banks of a distributed memory architecture.

A *thread molecule*, $(\!| e, \Sigma |\!)$, corresponds to one of the `main`-functions (see Section 2) where e is an *evaluation state* and Σ is a stack of (function-return) continuations as in Landin's SECD machine. As with queue molecules, we use the associative operator, $\bullet$, to separate elements on the stack and write ϵ for the empty stack. The syntax of evaluation states, e, is essentially the same as the syntax of expressions given in Figure 2, save that as noted above unbound variables do not occur but values may also be packet references, α. We write "$\{\vec{x}\}e$" for the continuation that binds variables $\vec{x}$ and continues as evaluation state e.

[5] There is no syntactic (compile-time) form of such values.

[6] If the offset is outside the domain of the packet the result is undefined.

Core language primitives:

$$[\![\{\vec{x}\}e_1]\!]_f \mid (\!|\texttt{let } \vec{y} = f(\vec{v}) \texttt{ in } e_2, \Sigma|\!) \rightsquigarrow [\![\{\vec{x}\}e_1]\!]_f \mid (\!|e_1\{\vec{v}/\vec{x}\}, (\{\vec{y}\}e_2) \bullet \Sigma|\!) \quad (\textit{user-call})$$
$$(\!|\texttt{let } x = v_1 \; op \; v_2 \texttt{ in } e, \Sigma|\!) \rightsquigarrow (\!|e\{v/x\}, \Sigma|\!) \text{ where } v = v_1 \; op \; v_2 \quad (\textit{basic-op})$$
$$(\!|\texttt{return } \vec{v}, (\{\vec{x}\}e) \bullet \Sigma|\!) \rightsquigarrow (\!|e\{\vec{v}/\vec{x}\}, \Sigma|\!) \quad (\textit{ret})$$
$$(\!|\texttt{if } \textit{true} \texttt{ then } e_1 \texttt{ else } e_2, \Sigma|\!) \rightsquigarrow (\!|e_1, \Sigma|\!) \quad (\textit{if}_1)$$
$$(\!|\texttt{if } \textit{false} \texttt{ then } e_1 \texttt{ else } e_2, \Sigma|\!) \rightsquigarrow (\!|e_2, \Sigma|\!) \quad (\textit{if}_2)$$

Packet manipulation:

$$\langle H \rangle \mid (\!|\texttt{let } x = \textit{new}(i) \texttt{ in } e, \Sigma|\!) \rightsquigarrow \langle H[\alpha \mapsto \textit{empty}_i] \rangle \mid (\!|e\{\alpha/x\}, \Sigma|\!) \quad (\textit{new})$$
where α is a fresh name

$$\langle H \rangle \mid (\!|\texttt{let } () = \textit{kill}(\alpha) \texttt{ in } e, \Sigma|\!) \rightsquigarrow \langle H \backslash \alpha \rangle \mid (\!|e, \Sigma|\!) \quad (\textit{kill})$$
$$\langle H \rangle \mid (\!|\texttt{let } x = \textit{read}(\alpha, i) \texttt{ in } e, \Sigma|\!) \rightsquigarrow \langle H \rangle \mid (\!|e\{H(\alpha)(i)/x\}, \Sigma|\!) \quad (\textit{read})$$
$$\langle H \rangle \mid (\!|\texttt{let } () = \textit{update}(\alpha, i, v) \texttt{ in } e, \Sigma|\!) \rightsquigarrow \langle H[\alpha \mapsto H(\alpha)[i \mapsto v]] \rangle \mid (\!|e, \Sigma|\!) \quad (\textit{update})$$

Queues:

$$\langle\!\langle \alpha \bullet Q \rangle\!\rangle_q \mid (\!|\texttt{let } x = q.\textit{deq}() \texttt{ in } e, \Sigma|\!) \rightsquigarrow \langle\!\langle Q \rangle\!\rangle_q \mid (\!|e\{\alpha/x\}, \Sigma|\!) \quad (\textit{deq})$$
$$\langle\!\langle Q \rangle\!\rangle_q \mid (\!|\texttt{let } () = q.\textit{enq}(\alpha) \texttt{ in } e, \Sigma|\!) \rightsquigarrow \langle\!\langle Q \bullet \alpha \rangle\!\rangle_q \mid (\!|e, \Sigma|\!) \quad (\textit{enq})$$

Fig. 5. Operational Semantics for PacLang

We write $init(P)$ to denote the initial state for program P. It consists of a function molecule, $[\![\{\vec{x}\}e]\!]_f$, for each function definition, $f(\vec{x})\{e\}$, in P; a thread molecule, $(\!|e_j, \epsilon|\!)$, for each definition $\texttt{main}_j()\{e_j\}$ in P; an empty queue molecule $\langle\!\langle \epsilon \rangle\!\rangle_q$ for each queue definition in P; and an empty heap $\langle\ \rangle$.

The semantics is given as a transition relation, $\rightarrow$, on states (multisets of molecules). For convenience, we define $\rightarrow$ in terms of an operator, $\rightsquigarrow$, which operates on only a subset of the molecules in a complete state:

$$\frac{M_1 \mid \ldots \mid M_n \rightsquigarrow M'_1 \mid \ldots \mid M'_n}{\Delta_1 \mid \ldots \mid \Delta_k \mid M_1 \mid \ldots \mid M_n \rightarrow \Delta_1 \mid \ldots \mid \Delta_k \mid M'_1 \mid \ldots \mid M'_n} \; (\rightsquigarrow)$$

where $\Delta_1, \ldots, \Delta_k$ are molecules that form part of the state, but are not of interest to the $\rightsquigarrow$-transition. Figure 5 gives transition rules for PacLang. Capture-free substitution of values $\vec{v}$ for variables $\vec{x}$ in expression e is written $e\{\vec{v}/\vec{x}\}$.

The outside world interacts with the program by placing network packets on the `receive` queue and removing network packets from the `transmit` queue. In reality these queues would be filled and emptied by hardware. However, to avoid extra semantic rules to express this, we can model a generic network interface as if it were two system-provided threads, one of which dequeues packets from `transmit` and frees them; the other of which constructs and allocates appropriate packets and enqueues them on `receive`.

4.1 Unique Ownership

Recall from Section 1 that the primary motivation for our type-system is to ensure that every packet in the heap is referenced by exactly one thread. We have argued that this *unique ownership* property makes practical the mapping

of a high-level program to a variety of diverse hardware architectures (see Section 1.1). In this section we define unique ownership formally.

Let, $pr(M)$, be the set of packet references, α, that occur in molecule M. (This is empty for function molecules, but generally non-empty for thread and queue molecules; $pr(\cdot)$ is not applied to the heap molecule.) We write $S_1 \uplus S_2$ to be the union of sets S_1 and S_2 under the condition that the elements of S_1 and S_2 are disjoint ($S_1 \cap S_2 = \emptyset$).

Definition 1 (Unique Ownership).

$$UniqueOwn(M_1 \mid \ldots \mid M_n \mid \langle H \rangle) \iff pr(M_1) \uplus \ldots \uplus pr(M_n) = \text{dom } H$$

Theorem 1 (Preservation of Unique Ownership). *If a program, P, types then any state, W, resulting from execution of P satisfies unique ownership:*

$$\vdash^{p} P \quad \wedge \quad init(P) \xrightarrow{*} W \quad \Rightarrow \quad UniqueOwn(W)$$

A proof of this theorem is presented in an accompanying technical report [6].

5 Case Study: An IPv4 Packet Forwarder

In this section we outline the PacLang implementation of the IP (version 4) uni-cast packet forwarding algorithm. The precise details of IP packet forwarding are not the focus of this section and, indeed, due to space constraints are not described here (the interested reader is referred to the IETF standards [13,14]). Instead, the purpose of the example is to outline the implementation of a realistic application, demonstrating that (*i*) packets are indeed treated as linear objects in real-life applications; and (*ii*) that the PacLang type-system is intuitive and easy to use.

Figure 6 shows the top-level structure of our PacLang IPv4 packet forwarder; we omit many of the function bodies, giving only their signatures. Function `eth_readAndClassify()` reads ethernet packets from the network interface and classifies them according to their type. Function `ip_lookup` performs a route lookup (on an externally defined routing table) returning both the next hop IP address and the network interface on which to forward the packet. (Note that since the external routing table component only contains `int`s, we do not have to worry about it breaking unique ownership.)

The program is divided into two separate threads (`main1` and `main2`). The `main1` thread sits in a tight loop, repeatedly calling the read/classify function. Packets which require ICMP processing are written to the `icmpQ` queue. The `main2` thread reads packets from `icmpQ` and performs the ICMP processing. The details of this thread are omitted. An example of a side-effecting function that manipulates packets is given by the `ip_dec_ttl` function which decrements the IP Time-To-Live field.

The PacLang code given here demonstrates that our linear type system naturally models the way packets flow through the system as a whole. Note

```
int eth_type(!packet p); // read type-field of ethernet frame
void eth_gen_arp_response(packet p); // generate response to ARP query
void eth_proc_arp_response(packet p); // process ARP response packet
void ip_local_packet(packet p); // send packet to local network stack
void ip_process_options(!packet p); // process IP options
void ip_transmit(packet p, int tx_interface, int ip_nexthop);
  // do MAC-level processing, get MAC address of next hop and transmit
int eth_rx_iface(!packet p); // which interface was packet from?

packetQueue icmpQ; // queue for packets which require ICMP processing

(int, int) ip_lookup(int dest_ip_addr);
     // lookup IP address and network interface for next-hop in route

void eth_readAndClassify() {
   packet p = receive.deq();
   switch (eth_type(p)) of {
      case ARP_QUERY: eth_gen_arp_response(p);
      case ARP_RES: eth_proc_arp_response(p);
      case IP: ip_forward(p);
      default: kill(p); }}

void ip_forward(packet p) {
   if (ip_packet_for_localhost(p)) ip_local_packet(p);
   else {
     (int nexthop, int iface) = ip_lookup(eth_ip_dest_addr(p));
     if (nexthop == 0) icmpQ.enq(p);
     else { if (iface==eth_rx_iface(p)) icmpQ.enq(p);
            else ip_forward_packet(p,iface,nexthop); }}}

void ip_forward_packet(Packet p, int iface, int nexthop) {
   if (ip_dec_ttl(p)) icmpQ.enq(p);
   else { ip_process_options(p);
          ip_transmit(p,tx_iface,ip_nexthop); }}}

bool ip_dec_ttl(!packet p) {
   if (p[ip_ttl]==0) return true;
   else { p[ip_ttl] = p[ip_ttl]-1; return false; }}

// thread 1 -- read, process, transmit loop:
void main1() { eth_readAndClassify(); main1(); }

// thread 2 -- process icmp packets:
void main2() { packet p = icmpQ.deq(); ... ICMP processing ...; main2(); }
```

Fig. 6. Top-level structure of an IPv4-over-ethernet packet forwarding engine

particularly that aliases (*!packet* types) allow us to express imperative operations on packets without requiring a packet's owning reference to be tediously threaded through all imperative operations (cf. the Clean language [8]).

Let us now consider compiling the PacLang IPv4 packet forwarder to the architecture shown in Figure 1.1. Due to PacLang's unique ownership property a compiler is free to choose between either (say) implementing both threads on a single IXP2400, implementing the queue using a shared memory; or placing `main1` on the first IXP2400, and `main2` on the second, passing packets by value between the two. In the latter case the queue itself could be located on either the first or second network processor and, further, as soon as a packet is transmitted from the first to the second IXP it can be immediately deallocated from the memory of the first. Although PacLang makes these tradeoffs very easy to

reason about, recall that this is not the case in the framework of traditional imperative languages (e.g. C/C++)—see Section 1.1.

6 Related Work

There has been a great deal of work in the area of linear types. Of this, our research is most closely related to Kobayashi's quasi-linear type system [12]. Aliases in our work (*!packet*) correspond to δ-uses in the quasi-linear framework. Similarly, owning references (of type *packet*) correspond to 1-use types. There are three major differences between our work and Kobayashi's. Firstly, we do not deal with higher-order functions: these add much of the perceived complication of linear types; hence this makes our type system much simpler and, we argue, more accessible to industrial software engineers. Secondly, we work with a concurrent language. Thirdly, whereas Kobayashi is primarily concerned with eliminating garbage collection, we use linear types to assist in the mapping of high-level specifications to distributed memory architectures.

Clarke et al. have proposed *ownership types* [5,4] for object-oriented languages. Each ownership type is annotated with a *context declaration*. The type system ensures that types with different declared contexts cannot alias. Ownership types have been applied to the detection of data races and deadlocks [3].

The `auto_ptr` template in the C++ standard library [15] exploits the notion of "unique ownership" *dynamically*. Suppose `p1` and `p2` are of type `auto_ptr`, then the assignment `p2 = p1` copies `p1` to `p2` as usual but also nulls `p1` so it cannot be used again. Similarly, if an `auto_ptr` is passed to a function it is also nulled. In this framework violation of the unique ownership constraint is only detected at run-time by a null pointer check. However, the widespread use of `auto_ptr`s in practical C++ programming suggests that working software engineers find the notion of object ownership intuitive and useful.

Further discussion of related work, including a comparison with *region*-based memory management approaches, is included in an associated technical report [6].

7 Conclusions and Future Work

We have formally defined the PacLang language and claimed, with reference to a case-study, that it is capable of specifying realistic packet processing applications. Our linear type system enforces a unique ownership property whilst allowing software engineers to program in a familiar style. We have argued that unique ownership greatly simplifies the compilation of a high-level program to a diverse range of NP architectures.

There are a number of ways in which PacLang must be extended if it is to be used on an industrial-scale. For example, the type system needs to be able to differentiate between different types of *packet* (e.g. IP packets, UDP packets etc.); support for structured types (such as records and tuples) and a module system are also required. However, much of this work amounts to engineering

rather than research effort. We believe that PacLang, as presented in this paper, captures the essence of packet-processing, providing a solid foundation for the development of an industrial-scale packet processing framework.

Our current work focuses on the implementation of a PacLang compiler which targets a variety of platforms, including Network Processors. This implementation effort will provide insights that can be fed back into the PacLang language design.

We hope that the ideas presented in this paper can be applied to the automatic partitioning of high-level code across multi-core architectures more generally (i.e. not just Network Processors). Since industrial trends suggest that such architectures will become more prevalent (as silicon densities continue to increase) we believe that this is an important topic for future research.

Acknowledgements. The authors wish to thank Tim Griffin and Lukas Kencl for their valuable comments and suggestions. We are grateful for the generous suppport of Microsoft Research, who funded the studentship of the first author.

References

1. Allen, J *et al.* PowerNP network processor: Hardware, software and applications. *IBM Journal of research and development 47*, 2–3 (3003), 177–194.
2. Berry, G., and Boudol, G. The chemical abstract machine. *Theoretical Computer Science 96* (1992), 217–248.
3. Boyapati, C., Lee, R., and Rinard, M. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-02)* (November 2002), ACM Press.
4. Clarke, D., and Wrigstad, T. External uniqueness is unique enough. In *Proceedings of the 17th European Conference for Object-Oriented Programming (ECOOP)* (July 2003), vol. 2743 of *LNCS*, Springer-Verlag.
5. Clarke, D. G., Potter, J. M., and Noble, J. Ownership types for flexible alias protection. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)* (New York, 1998), vol. 33:10 of *ACM SIGPLAN Notices*, ACM Press, pp. 48–64.
6. Ennals, R., Sharp, R., and Mycroft, A. Linear types for packet processing (extended version). Tech. Rep. UCAM-CL-TR-578, University of Cambridge Computer Laboratory, 2004.
7. Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. The essence of compiling with continuations. In *Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993*, vol. 28(6). ACM Press, New York, 1993, pp. 237–247.
8. Hilt High Level Software Tools B.V. Concurrent clean language report. Available from: `ftp://ftp.cs.kun.nl/pub/Clean/Clean20/doc/CleanRep2.0.pdf`.
9. Intel Corporation. Intel IXDP2400 & IXDP2800 advanced development platforms product brief. Available from: `http://www.intel.com/design/network/prodbrf/ixdp2x00.htm`.

10. Intel Corporation. Intel IXP2400 Network Processor: Flexible, high-performance solution for access and edge applications.
Available from: `http://www.intel.com/design/network/papers/ixp2400.htm`.
11. Johnson, E. J., and Kunze, A. *IXP-1200 Programming: The Microengine Coding Guide for the Intel IXP1200 Network Processor Family.* Intel Press, 2002.
12. Kobayashi, N. Quasi-linear types. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas* (New York, NY, 1999), pp. 29–42.
13. Postel, J. Internet Protocol RFC 791. Internet Engineering Task Force, September 1981. Available from: `http://www.ietf.org/rfc.html`.
14. Postel, J. Internet Control Message Protocol, RFC 792. Internet Engineering Task Force, September 1981. Available from: `http://www.ietf.org/rfc.html`.
15. Stroustrup, B. *The C++ programming language (third edition).* Addison-Wesley.
16. Turner, D. N., Wadler, P., and Mossin, C. Once upon a type. In *Functional Programming Languages and Computer Architecture* (1995), ACM Press.
17. Yavatkar, R., and H. Vin (eds.). *IEEE Network Magasine. Special issue on Network Processors: Architecture, Tools, and Applications 17*, 4 (July 2003).