# Formalizing Address Spaces with application to Cuda, OpenCL, and beyond


Benedict R. Gaster
Advanced Micro Devices, 1 AMD, Sunnyvale, CA, USA
benedict.gaster@amd.com

Lee Howes
Advanced Micro Devices, 1 AMD, Sunnyvale, CA, USA
lee.howes@amd.com



## Abstract

Cuda and OpenCL are aimed at programmers developing parallel applications targeting GPUs and embedded micro-processors. These systems often have explicitly managed memories exposed directly though a notion of disjoint address spaces. OpenCL address spaces are based on a similar concept found in Embedded C. A limitation of OpenCL is that a specific pointer must be assigned to a particular address space and thus functions, for example, must say which pointer arguments point to which address spaces. This leads to a loss of composability and moreover can lead to implementing multiple versions of the same function. This problem is compounded in the OpenCL C++ variant where a class' implicit *this* pointer can be applied to multiple address spaces.

Modern GPUs, such as AMD's Graphics Core Next and Nvidia's Fermi, support an additional generic address space that dynamically determines an address' disjoint address space, submitting the correct load/store operation to the particular memory subsystem. Generic address spaces allow for dynamic casting between generic and non-generic address spaces that is similar to the dynamic subtyping found in objected oriented languages. The advantage of the generic address space is it simplifies the programming model but sometimes at the cost of decreased performance, both dynamically and due to the optimization a compiler can safely perform.

This paper describes a new type system for inferring Cuda and OpenCL style address spaces. We show that the address space system can be inferred. We extend this base system with a notion of generic address space, including dynamic casting, and show that there also exists a static translation to architectures without support for generic address spaces but comes at a potential performance cost. This performance cost can be reclaimed when an architecture directly supports generic address space.


## 1. Introduction

Address spaces play a fundamental role in description of data locality in programming languages, allowing the developer to explicitly manage where data lives during program execution. Originally developed as a generic extension to the Embedded C [12] variant of ANSI C, address spaces have recently gained popularity in programming languages for General Purpose Graphics Processing Units (GPGPU). In particular, Nvidia's Cuda [21] has support for disjoint address spaces as a type modifier, while Khronos' Open Compute Language (OpenCL) [22] formalizes them as type qualifiers as a variant of Embedded C, shown diagrammatically in Figure 1. A work-item is an instance of kernel at each projection point within a 3D iteration space, with access to its own *private* memory, a *local* memory that is shared between a collection of work-items (called a work-group), and finally a *globally* visible memory shared between all concurrently executing work-items. Each address space is disjoint and is assumed not to overlap.

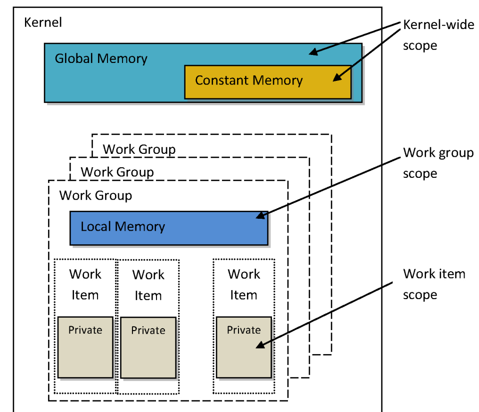


**Figure 1.** Abstract memory model defined by OpenCL

As an example consider the following code that scales a vector ($A$) by a constant(s) outputting a vector ($C$)[1]:

```
kernel void vscale(global int * C, global int * A,
                              const global int * S)
{
   C[get_global_id(0)] =
     A[get_global_id(0)] * S[get_group_id(0)];
}
```

The implication of OpenCL address spaces are that every pointer must be associated with an address space. The drawback with this approach, relaxed somewhat in Cuda's model, is the inability to parameterize over address spaces, i.e. define parametric polymorphic

[1] The function `get_global_id(size_t)` returns the projection within the 3D iteration space that the kernel is being executed over, the argument (0,1, or 2) selects the dimension.


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.



*GPPGU-6,* March 16 2013, Houston, TX, USA



Copyright © 2013 ACM 978-1-4503-2017-7/13/03. . . $15.00

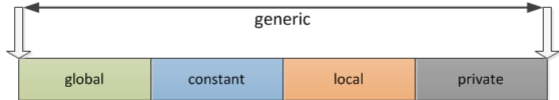


**Figure 2.** Generic address space

functions that are implicitly parametrized by address spaces. For example, consider a simple function to scale a value:

```
int scale(global int * A, global int * S);
```

It is easy to adapt the kernel *vscale* above to call *scale*. However, a typical optimization would be to move *S* into optimized on chip memory accessed via the read only memory segment *constant*:

```
kernel void vscale(global int * C, global int * A,
                   constant int * S)
{
   C[get_global_id(0)] = scale(&A[get_global_id(0)],
                               &S[get_group_id(0)]);
}
```

Of course, this is will no longer type check as the type of *S* no longer matches *scales* second argument. To address this limitation we introduce an additional address space *generic*. (Note we assume *generic* to be the default address space and as such it can be elided in practice.) *scale* would be prototyped as:

```
int scale(generic int * A, generic int * B);
```

The *generic* address space defines a single address space that subsumes all others, as depicted in Figure 2. The particular placement of the different memory spaces within *generic* is an implementation detail that is independent of our definition and use of generic, i.e. we could have equally have placed *global* at the far right of the address map and seen no semantic difference.

In many cases it is straightforward to infer a specific address space instance in place of a generic one, using a modified version of the Hidley-Milner type inference algorithm [4, 10, 18]. However, in general it is not possible because a value within a generic address space may take on the type of multiple address spaces over its life time. For example, consider the following code that assigns a pointer in the global address space to each even work-item and a pointer in local address space to each odd work-item:

```
void foo(int *);

kernel void bar(
   global int *g, local int *l)
{
  generic int * tmp;

  if (get_global_id(0) % 2) {
     tmp = g;
  }
  else {
     tmp = l;
  }

  foo(tmp);
}
```

In this case to preserve the single code base for all work-items the type of *tmp* needs to allow for the alternative address spaces *global* and *local*. Such alternatives are common in type systems and are known as variant or sum types, i.e. an alternative type of *tmp* is:

```
global + local int *;
```

with one alternative indicating that a value is in the *global* address space and the other in the *local* address space.

Unfortunately, while the introduction of variant address spaces goes part of the way to providing types for generic address spaces, it requires that a particular address space component be uniquely determined at compile-time. The implication is that such a system does not support both variant address spaces and parametric polymorphism over address spaces, i.e. it would fail to provide a principal type for *foo* in the above example. What is needed is a way of combining generic address spaces with variant address spaces. The type system proposed in this paper is just such a system.

Before describing our system in detail we first consider the runtime implementation of such a system. In particular, due to the disjoint nature of address spaces, an implementation is free to support different load/store operations for different address spaces. An implication of such hardware targets is a compiler must be able to determine, at compile-time, a unique address space. However, not all targets have this limitation and in fact Nvidia's Fermi [20] and AMD's Graphics Core Next (GCN) [17] both support a notion of generic address space, providing a mapping conceptually similar to that of Figure 2.

Even with hardware support for generic address spaces it may be beneficial to emit specialized (i.e. unique address space) loads and stores for performance or power considerations. For example, both Fermi and GCN provide specialized load and store operations.

### 1.1 This paper

The type system described in this paper combines the notion of generic address spaces with variant address spaces to provide a practical type system for languages such as Cuda [21] and OpenCL [22]. In particular, it supports polymorphic (i.e. generic) address spaces, extensibility (the ability to add or remove address spaces from a type), type inference, down casting to and from generic address spaces, and compilation. The type system is an application of qualified types, extended to deal with a general concept of polymorphic address spaces. Positive information about which address spaces are expected is captured in a given address space variable, called a *row*, using row extension, while negative information is reflected by the use of predicates.

The most obvious benefit of this is that we can adapt results and properties from the general framework of qualified types—such as the type inference algorithm and the compilation method—without having to go back to first principles. The result is a considerable simplification of both the overall presentation and of specific proofs.

One important aspect of our system is that while formally defined it has very practical implications. In particular, we contacted and worked with AMD's OpenCL team to implement a modified version of our algorithm in their implementation of OpenCL C++, a C++ language extension for programming OpenCL devices. We expect that our algorithm will form the foundation for specifying generic address spaces within OpenCL C too and could be used in a Cuda compiler flow to generate more refined address space access information.

This paper makes a number of novel and important contributions:

- We develop a theory of address spaces that is applicable to Cuda, OpenCL and other Embedded C variants. Including, formalizing address space type inference and support for generic (polymorphic) address spaces.
- We show that our system can be applied to the dynamic generic address space of architectures such as AMD's GCN and Nvidia's Fermi.

| | |
|---|---|
| $a$ ::= $global \mid local \mid private$ | address spaces |
| $t$ ::= $int$ | interger types |
| $\mid t*$ | pointer types |
| $\mid ASpace\ \{a \mid r\}\ t$ | address space qualifed type |
| $p$ ::= $t\ x$ | parameter/field specifications |
| $e$ ::= $x \mid c \mid e(\vec{e})$ | variables, constants, applications |
| $\mid e$ op $e$ | binary operations |
| $vd$ ::= $p = e$ | value declarations |
| $s$ ::= $e; \mid vd;$ | expressions, declarations |
| $\mid *a\ e;$ | indirect load |
| $\mid$ return $e;$ | return from function |
| $\mid x = e;$ | assignments |
| $\mid *x = e;$ | indirect assignments |
| $\mid$ if $(e)$ $\{\ \vec{s}\ \}$ else $\{\ \vec{s}\ \}$ | conditionals |
| $\mid$ while $(e)$ $\{\ \vec{s}\ \}$ | while loop |
| $d$ ::= $vd$ | value declarations |
| $\mid t\ x(\vec{p})\ \{\vec{s}\}$ | function declarations |
| $\mid$ kernel void $x(\vec{p})\ \{\vec{s}\}$ | kernel declarations |
| $tl$ ::= $\vec{d};$ | declarations |

**Figure 3.** The MiniAS concrete syntax. We assume non-terminals $x$ for identifiers, $c$ for constants, and $r$ for row variables. The non-terminal *op* ranges over binary operations, e.g. $+, -$, etc.

- We show that the theory of qualified types has application outside of its traditional application area, that of functional programming languages such as Haskell [16] and Habit [19].
- We have implemented a prototype compiler, in the functional language Haskell [16], that can be used by other compiler writers to develop practical implementations of their own. As noted above AMD's OpenCL compiler team have already done just this in practice, and is the foundation of their OpenCL C++ implementation found in the product APP SDK 2.7.

A note to the reader; this paper describes a formal foundation for address space used in languages such as Cuda and OpenCL, including describing a system for inferring address space usage when emitted by the programmer. This paper does not describe particular syntactic constructs for extending languages like Cuda and OpenCL with generic address spaces, however, a companion paper, also at this conference, describes an application of this model to OpenCL C++l [8]. This companion paper also includes an evaluation of OpenCL C++'s key features, including generic address spaces. It is the intention that these two papers be read together.

The remaining sections of this paper are as follows: Section 2 provides a general overview of our new type system, with a more detailed formal presentation in Section 3. This is followed by discussion of type inference in Section 4 and compilation in Section 5. Section 6 discusses how the base system can be extended to support casting to and from generic address spaces. Section 7 discusses related and previous work. Finally, Section 8 concludes with a discussion of future work.

## 2. Overview

Since Cuda and OpenCL C are extensions to C++ and C, respectively, both of which are too complex for a concise formal definition, we concentrate here on a subset of these languages that reflects essential aspects of the extensions.

With out loss of generality we further choose to concentrate on the extended C subset of C as this simplifies the type system considerably. We note that it is straightforward to extend out system with class subtyping, handled with bounded polymorphism [2, 3], and handle recursive template subtyping with bounded polymorphism [2, 3]. We call this language *MiniAS*.

The abstract syntax for MiniAS programs is given in Figure 3.

### 2.1 Basic operations

Address space types are defined in terms of rows, and these are constructed by extension, starting from the empty row, $\{\}$. It is convenient to use the following abbreviations for rows:

$$\begin{aligned} \{a_1, ..., a_n \mid r\} &= \{a_1 \mid ...\{a_n \mid r\}...\} \\ \{a_1, ..., a_n\} &= \{a_1 \mid ...\{a_n \mid \{\}\}...\} \end{aligned}$$

Intuitively, an address space of type $ASpace\ \{a \mid r\}\ \tau *$ is a variant (or union) whose component $a$ implies that the resulting pointer type $\tau *$ can be in that address space, and whose component $Aspace\ r$ ranges over some still to be determined address spaces. The basic operations for address spaces are:

- Definition (injection): to define a pointer, with initializer, with an address space:

$$\begin{aligned} \tau * x &= _ :: (r \backslash a) \Rightarrow \tau \to ASpace\ \{a \mid r\}\ \tau * \\ a\,\tau * x &= _ :: (r \backslash a) \Rightarrow \tau \to ASpace\ \{a \mid r\}\ \tau * \end{aligned}$$

- Assignment (injection): to perform an assignment of a pointer in the same address space

$$_ = _ :: (r \backslash a) \Rightarrow ASpace\ \{a \mid r\}\ \tau * \to ASpace\ \{a \mid r\}\ \tau * \to ASpace\ \{a \mid r\}\ \tau *$$

- Assignment (Embedding): to perform an assignment of a pointer with a new address space:

$$_ = _ :: (r \backslash a) \Rightarrow ASpace\ r\ \tau * \to ASpace\ \{a \mid r\}\ \tau * \to ASpace\ \{a \mid r\}\ \tau *$$

- De-reference: to perform a de-reference from pointer address[2]:

$$ld(_) :: ASpace\{a\}\ \tau * \to \tau$$

$$ld_a(_) :: (r \backslash a) \Rightarrow (ASpace\{a\}\ \tau * \to \tau) \to ASpace\ \{a \mid r\}\ \tau \to \tau$$

$$\begin{aligned} ld(_,_,_,_,_) :: \ & (ASpace\{global\}\ \tau * \to \tau) \\ & \to (ASpace\{local\}\ \tau * \to \tau) \\ & \to (ASpace\{private\}\ \tau * \to \tau) \\ & \to ASpace\ r\ \tau * \\ & \tau \end{aligned}$$

The empty address space, $\{\}$, is the only value of type $ASpace\{\}$. Predicates are useful for the formal system described below but are not required to be written in MiniAS programs. When is clear from context that an address space is monomorphic, i.e. a unique address space, we will write:

```
global int * x = ...;
local int * y = ...;
*x = *y;
```

when formally we would have written:

```
ASpace { global } int * x = ...;
ASpace { local } int * y = ...;
st_global(x, ld_local(y));
```

In the case of polymorphic address spaces (i.e. row variables) we will just elide the address space altogether, e.g.:

[2] We elide the store case as it is defined in a similar fashion and adds little.

```
ASpace r int * var = 0;
```

will be written as

```
int * var;
```

dropping the zero initializer too.

The interesting case is de-reference, when the particular address space is not known at compile time and thus must work for any valid address space. For example, consider then following:

```
kernel void x(
  global  * int g,
  local  * int l,
  int value)
{
   int * var = 0;
   if (value % 2) {
      var = g;
   }
   else {
      var = l;
   }

   *g = *var;
}
```

In general, is not possible to know the address space for the dereference, $*var$, and so the dereference operation must be able to perform a load from any address space. Filling in all the annotations the example would be written as:

```
kernel void x(
  ASpace { global }  * int g,
  ASpace { local }  * int l,
  int value)
{
   ASpace r int * var = 0;
   if (value % 2) {
      var = g;
   }
   else {
      var = l;
   }

   store_global(g,
     ld(var, ld_global, ld_local, ld_private);
}
```

The example provides implementations, i.e. ld_aspace, for all possible address spaces and thus is total, i.e. will not cause an unexpected load from address space error.

### 2.2 Implementation details

The implementation of address spaces must select, in the fully generic case at runtime, the load or store instruction that matches an individual address space. To select a particular load $load_aspace$ from an address space $a$, we need to know the address space ID representing the value $a$. Each address space is assigned an integer ID defined as follows[3]:

$$
\begin{aligned}
global &= 0 \\
local &= 1 \\
private &= 2
\end{aligned}
$$

MiniAS programs without generic address load/stores, i.e. standard OpenCL 1.2 programs, only contain load/stores whose address space is known, and hence the full type of $a$, is known at compile-time.

In the more general case of generic address spaces it is not necessary to know the address space (ID) for every load and store at compile-time; instead, we treat unknown offsets as implicit parameters whose values will be supplied at run-time when the full types of the load/stores concerned are known. Intuitively, load/stores are implemented as a jump table, where the ID provides the index into the jump table, selecting a specific load/store. This is essentially the compilation method of Gaster and Jones [9]. If for a moment we forget about typing issues, then the $load(_, _, _, _, _)$ could be implemented as:

```
load(idx, gld, lld, pld, addr)
{
  switch idx {
    case 0:
      return gld(addr);
    case 1:
      return lld(addr);
    case 2:
      return pld(addr);
  }
}
```

Of course, there are run-time overheads in passing offset values as extra parameters. However, an attractive feature of our system is these costs are only incurred when the extra flexibility of generic address spaces is required. Moreover, an architecture that supports generic address spaces directly, e.g. Nvidia's Fermi, can simply elide the additional parameters and jump-tables, issuing a single load or store instruction. Each predicate $r \backslash a$ in the type of a function signals the need for an extra run-time parameter to specify the address-space used to determine the particular load/store. This one single extra piece of information is all that is needed to implement the full set of address space operations.

The type checker gathers and simplifies the predicates generated by each use of an operator on address spaces. For example, the derived type, for the load of $x$, in the following:

```
global int * x = ...;
... *x ...;
```

will generate a single constraint, $\{\} \backslash global$. Predicates, like this, involving rows whose structure is known at compile-time, are easily discharged by calculating the appropriate address space ID. Obviously, a compiler can use this information to produce efficient code by inlining and specializing to emit a specific load instruction, for the corresponding address space. It is possible to show that for all MiniAS programs that use only explicit address spaces, i.e. OpenCL 1.2 programs, then all predicates can be statically determined and thus discharged at compile time. We return to this and additional static properties during the formalization of the model.

Predicates that are not discharged within a section of code will, instead, be reflected in the type assigned to it. For an implementation not supporting generic address spaces in hardware it is possible to easily define rules that restrict generic address spaces for function arguments and variable definitions to be defined only for a single address-space instance. For example, a compiler might reject types which contain multiple predicates for the same row variable, but allow functions whose arguments are generic address spaces. This would allow:

```
kernel void foo(int * x)
{
   ...
}
```

[3] This mapping is just one possible mapping.

```
global int * g = ...;
local int * l = ...;

foo(g);
foo(l);
```

while disallowing:

```
kernel void bar(global int * g, local int * l)
{
   int * a;

  if (...) {
     a = g;
  } else {
     a = l;
  }
}
```

A consequence of these restrictions is that it is straightforward to define a translation, based on the notion of simplification [13], to specialize, at compile time, all predicated function calls, to a MiniAS program that is guaranteed to contain no discharged predicates[4].

## 3. Formal presentation

MiniAS's type system is based on Jones' theory of qualified types [13] adapting the notion of subtyping for records and variants developed by Gaster and Jones [9].

### 3.1 Kinds

| | |
|---|---|
| $k ::= *$ | the kind of all types |
| $\mid row$ | the kind of rows |
| $\mid aspace$ | the kind of address spaces |
| $\mid k_1 \rightarrow k_2$ | function kinds |

Intuitively the kind $k_1 \rightarrow k_2$ represents the constructors that take something of kind $k_1$ and return something of kind $k_2$. The $row$ kind is from Gaster and Jones' system [9]. The $aspace$ kind represents the kind of address spaces and is new to the system presented in this paper.

### 3.2 Types and constructors

For each kind $k$, we have a collection of construtors $C^k$ (including variables $\alpha^k$ of kind $k$:

| | |
|---|---|
| $C^k ::= X^k$ | constants |
| $\mid \alpha^k$ | variables |
| $\mid C^{k' \rightarrow k} C^{k'}$ | applications |
| $\tau \ ::= C^*$ | types |

The usual collection of types, represented here by the symbol $\tau$, is just the constructors of kind $*$. For the purposes of this paper, we assume that the set of constant constructors includes at least the following, writing $X ::^k$ to indicate the kind $k$ associated with each constant $X$ :

| | | |
|---|---|---|
| $\rightarrow$ | $:: * \rightarrow * \rightarrow *$ | function space |
| $\{\}$ | $:: row$ | empty row |
| $\{_ \mid _\}$ | $:: aspace \rightarrow row \rightarrow row$ | extension, for each address space |
| $ASpace$ | $:: row \rightarrow * \rightarrow *$ | address space construction |

- The result of applying the function space constructor $\rightarrow$ to two types $\tau$ and $\tau'$ is the type of functions from $\tau$ to $\tau'$, and is written as $\tau \rightarrow \tau'$ in more conventional notation. Technically MiniAS functions are uncurried (i.e. of the form $(\tau_1, ..., \tau_n) \rightarrow \tau$) but for ease of description we will use curryed notation (i.e. $\tau_1 \rightarrow ... \rightarrow \tau_n \rightarrow \tau$).
- The result of applying the $ASpace$ constant to the empty row $\{\}$ of kind $row$ and some type $\tau *$ is the type $Aspace\ \{\}\ \tau *$ of kind $*$.
- The result of applying an extension constructor $\{_ \mid _\}$ to a type $\tau$ and a row $r$ is a row, usually written as $\{a \mid r\}$, obtained by extending $r$ with a an address space $a$. Note that we include an extension constructor for each different address space $a$.

[4] This restriction is effectively Haskell's monomorphism restriction [16].

The kind system is used to ensure that type expressions are well-formed. While it is sometimes convenient to annotate individual constructors with their kinds, there is no need in practice for a programmer to supply these annotations. Instead, they can be calculated automatically using a simple kind inference process [14].

We consider two rows to be equivalent if they include the same address spaces, regardless of the order in which they are listed. This is described formally by the equation:

$$\{a, a' \mid r\} = \{a', a \mid r\}$$

For the purposes of later sections, we define a membership relation, $a \in r$, to describe when a particular address space $a$ appears in a row $r$:

$$a \in \{a \mid r\} \qquad \frac{a \in r}{a \in \{a' \mid r\}} [a \neq a']$$

and a restriction operation, $r - a$ , that returns the row obtained from $r$ by deleting the address space $a$:

$$\{a \mid r\} - a = r$$
$$\{a' \mid r\} - a = \{a' \mid r - a\}$$

It is easy to prove that these operations are well-defined with respect to the equality on constructors, and to confirm intuitions about their interpretation by showing that, if $a \in r$, then $r = \{a \mid r - a\}$.

### 3.3 Predicates

The syntax for rows allows examples like $\{a, a\}$ where the address space $a$ appears in more than one field. Clearly, we do not want an address space to appear twice and some additional mechanisms are needed to enable us to specify that a type of the form $ASpace\{a \mid r\}$, for example, is only valid if the row $r$ does not also contain $a$. We achieve this using the lacks predicate of Gaster and Jones' [9]:

$$\pi :: C^{row} \backslash a \quad \text{predicates}$$

Intuitively, the predicate $r \backslash a$ can be read as an assertion that the row $r$ does not contain the address space a. More precisely, we explain the meaning of predicates using the entailment relation defined in Figure 4. A derivation of $P \models \pi$ from these rules can be understood as a proof that, if all of the predicates in the set $P$ hold, then so does $\pi$. It is easy to prove that the relation $\models$ is well-defined with respect to equality of constructors.

$$P \cup \{\pi\} \models \pi \qquad \frac{P \models r \backslash a \quad a \neq a'}{P \models \{a' \mid r\} \backslash a} \qquad P \models \{\} \backslash a$$

**Figure 4.** Predicate entailment for rows.

$$\frac{}{P|\Gamma \vdash ; : \texttt{void}}\,[\textsc{Empty}] \quad \frac{P|\Gamma \vdash e : \tau \quad P|\Gamma \vdash z : \tau'}{P|\Gamma \vdash \texttt{return}\, e; z : \tau}\,[\textsc{Ret}] \quad \frac{(x : \sigma) \in \Gamma}{P|\Gamma \vdash x : \sigma}\,[\textsc{Var}] \quad \frac{P|\Gamma \vdash e : \tau' \quad P|\Gamma \vdash z : \tau}{P|\Gamma \vdash e; z : \tau}\,[\textsc{Expr}]$$

$$\frac{P|\Gamma \vdash e : \tau \quad P|\Gamma, v : \tau \vdash z : \tau'}{P|\Gamma \vdash \ \tau\, v = e; z : \tau'}\,[\textsc{VDecl}] \quad \frac{P|\Gamma \vdash e : \texttt{bool} \quad P|\Gamma \vdash z_0; z : \tau \quad P|\Gamma \vdash z_1; z : \tau}{P|\Gamma \vdash \texttt{if}\ (e)\ \{z_0\}\ \texttt{else}\ \{z_1\}; z : \tau}\,[\textsc{If}]$$

$$\frac{v : \tau \in \Gamma \quad P|\Gamma, v : \tau \vdash e : \tau \quad P|\Gamma, v : \tau \vdash z : \tau'}{P|\Gamma \vdash \ v = e; z : \tau'}\,[\textsc{VAssign}] \quad \frac{P|\Gamma \vdash e : \texttt{bool} \quad P|\Gamma \vdash z : \texttt{void} \quad P|\Gamma \vdash z' : \tau'}{P|\Gamma \vdash \texttt{while}\ (e)\ \{z\}; z' : \tau'}\,[\textsc{While}]$$

$$\frac{P|\Gamma \vdash e : \pi \Rightarrow \rho \quad P \models \pi}{P|\Gamma \vdash e : \rho}\,[\Rightarrow E] \quad \frac{P, \pi|\Gamma \vdash e : \rho}{P|\Gamma \vdash e : \pi \Rightarrow \rho}\,[\Rightarrow I]$$

$$\frac{P|\Gamma \vdash e : \forall\alpha.\sigma}{P|\Gamma \vdash e : [\tau/\alpha]\sigma}\,[\forall E] \quad \frac{P|\Gamma \vdash e : \sigma \quad P|\Gamma \vdash e : \alpha \notin TV(\Gamma) \cup TV(P)}{P|\Gamma \vdash e : \forall\alpha.\sigma}\,[\forall I]$$

$$\frac{P|\Gamma, v_i : \tau_i \vdash e : b : \tau}{P|\Gamma \vdash \tau\, x(v_i)\ \{b\} :\to (t_i) \to \tau}\,[\textsc{Fcn}] \quad \frac{(x \to (t_i) \to \tau) \in \Gamma \quad P|\Gamma \vdash e_i : \tau_i}{P|\Gamma \vdash x(e_i) : \tau}\,[\textsc{App}]$$

$$\frac{P_i|\Gamma \vdash^W \ f_i(v_i)\{b_i\} : \sigma_i \quad P'|\Gamma_{f_i}, f_i : \sigma_i) \vdash^W \ b' : \tau}{P'|\Gamma \vdash^W \ f_i(v_i)\{b_i\}\ \texttt{main}()\{b'\} : \tau}\,[\textsc{Program}]$$

**Figure 5.** MiniAS Typing Rules

### 3.4 Typing rules

Following Damas and Milner [4], we distinguish between the simple types $\tau$, described above, and type schemes, $\sigma$, described by the grammar below:

| | |
|---|---|
| $\sigma ::= \rho \mid \forall\alpha.\rho$ | type schemes |
| $\rho ::= \tau \mid \pi \Rightarrow \tau$ | qualified types |

For simplicity of presentation and due to the fact that MiniAS does not support general polymorphic types, we restrict our presentation to type schemes with type variables of kind $row$.

Restrictions on the instantiation of universal quantifiers, and hence on polymorphism, are described by encoding the required constraints as a set of predicates, $P$, in a qualified type of the form $P \Rightarrow \tau$. The set of free type variables in a object $X$ is written as $TV(X)$.

The syntax for our term language is that of MiniAS, defined in Figure 3. Each of the address-space load and store operations is assigned a closed type scheme, $\sigma_{ld/st}$. The typing rules are presented in Figure 5.

## 4. Type inference

This section provides a formal presentation of a type-inference algorithm for inferring address space usage. The most important feature is our adaptation of Gaster and Jones' [9] inserters for address spaces, to account for non-trivial equalities between row expressions during unification.

### 4.1 Unification and insertion

Unification is a standard tool in type inference, and is used, for example to ensure that the formal and actual address space parameters of a function have the same type. Formally, a substitution is $S$ is a *unifier* of constructors $C, C' \in C^k$ if $SC = SC'$, and is a *most general unifier* of $C$ and $C'$ if every unifier of these two constructors can be written in the form $RS$, for some substitution $R$.

$$(id) \qquad C \overset{id}{\sim} C$$

$$(bindL) \qquad \alpha \overset{[C/\alpha]}{\sim} C \qquad \alpha \notin TV(C)$$

$$(bindR) \qquad C \overset{[C/\alpha]}{\sim} \alpha \qquad \alpha \notin TV(C)$$

$$(apply) \qquad \frac{C \overset{U}{\sim} C' \quad UD \overset{U'}{\sim} UD'}{CD \overset{U'U}{\sim} C'D'}$$

$$(row) \qquad \frac{a \overset{I}{\in} r' \quad Ir \overset{U}{\sim} (Ir' - a)}{\{a \mid r\} \overset{UI}{\sim} r'}$$

**Figure 6.** Kind-perserving unification

The rules in Figure 6 provide an algorithm for calculating unifiers, writing $C \overset{U}{\sim} C'$ for the assertion that $U$ is a unifier of the constructors $C, C' \in C^k$. The first three rules are standard [25], and are even suitable for unifying to row expressions that list exactly the same components with exactly the same ordering in each. But the forth rule, (row), is needed to deal with the more general problems of row unification.

To understand how this rule works, consider the task of unifying two rows $\{a \mid r\}$ and $\{a' \mid r'\}$, where $a$, $a'$ are distinct address spaces, and $r$, $r'$ are distinct row variables. Our goal then is to find a substitution $S$ that:

$$\begin{aligned} \{a \mid Sr\} &= S\{a \mid r\} \\ &= S\{a' \mid r'\} \\ &= \{a' \mid Sr'\} \end{aligned}$$

Clearly, the row on the left includes an $a$ address space, while the last row on the right include an $a'$ address space. If these two types are to be equal, then we must choose the substitution $S$ so that it will 'insert' the missing fields into the two rows $r'$ and $r$, respectively. In this particular case. then we can choose:

$$S = [\{a' \mid r''\}/r, \{a \mid r''\}/r']$$

where $r''$ is a new type variable.

More generally, we will say that a substitution $S$ is an *inserter* of $a$ into $r \in C^{row}$ if $a \in Sr$. $S$ is a *most general inserter* of $a$ into $r$ if ever such an inserter can be written in the form $RS$, for some substitution $R$. The rules in Figure 7 define an algorithm for calculating inserters of $a$ into $r \in C^{row}$.

$$(idVar) \qquad a \overset{[\{a \mid r'\}/r]}{\in} r \qquad r' \; new$$

$$(inTail) \qquad \frac{a \overset{I}{\in} r \quad a \neq a'}{a \overset{I}{\in} \{a' \mid r\}}$$

$$(inHead) \qquad a \overset{id}{\in} \{a \mid r\}$$

**Figure 7.** Kind-perserving insertion

The important properties of unification and insertion—both soundness and completeness—are captured in the following result:

THEOREM 4.1. *The unification (insertion) algorithm defined by the rules in Figure 6 (Figure 7) calculates most general unifiers (inserters) whenever they exist. The algorithm fails precisely when no unifier (inserter) exists.*

The proof is a straightforward variant of that given by Gaster [6], describing his system of records and variants. It is important to note that the unification algorithm is simplified, when compared to Gaster's original work, due to address space components not being labeled.

### 4.2 A Type inference algorithm

Given the unification algorithm described in the previous section, we can use an extended version of the type inference algorithm of qualified types [14] as a type inference algorithm for the type system presented in this paper. The definition of the algorithm is given in Figure 8. Following Rémy [24], these rules can be understood as an attribute grammar; in each typing judgement $P|T\Gamma \vdash^{W} e : \tau$, the type assignment $\Gamma$ and the term $e$ are inherited attributes, while the predicate assignment $P$, type $\tau$, and substitution $T$ are synthesized. The $(Program)^{W}$ rule uses an auxiliary function to calculate the generalization of a qualified type $\rho$ with respect to a type assignment $\Gamma$. This is defined as follows:

$$Gen(\Gamma, \rho) = \forall \alpha_i . \rho, where \{\alpha_i\} = TV(\rho) \backslash TV(\Gamma).$$

In general the rules are straightforward modifications of the original typing rules given in Figure 5 for type inference. For example, the rule $(IF)$ has additional hypothesis that perform unification on the inferred types of the condition and two alternatives, and applications of the synthesized substitutions, but otherwise is the same as the original typing rule. As such the type inference algorithm is both sound and complete with respect to the original typing rules.

THEOREM 4.2. *The algorithm described by the rules in Figure 8 can be used to calculate the principal type for a given declaration $d$ under the assumptions $\Gamma$. The algorithm fails precisely when there is no typing for $d$ under $\Gamma$.*

The proof is again straightforward and follows directly from the earlier work of Gaster [6] and more generally Jone's system of Qualified Types [13].

## 5. Compilation

In previous sections we described informally how programs involving operations on address spaces can be compiled and executed using a language that adds extra parameters to supply appropriate offsets. This section shows how this process can be formalized, including the calculation of address space IDs.

### 5.1 Compilation by translation

In the general treatment of qualified types [13], programs are compiled by translating them into a language that adds extra parameters to supply *evidence* for predicates appearing in the types of the values concerned. The whole process can be described by extending the typing rules to use judgements of the form:

$$P|\Gamma \vdash e \leadsto e' : \sigma$$

which include both the original source term $e$ and a possible translation $e'$. A further change here is the switch from predicate sets to predicate assignments; the symbol P used above represents a set of pairs $(v : \pi)$ in which no variable $v$ appears twice. Each variable $v$ corresponds to an extra parameter that will be added during compilation; $v$ can be used whenever evidence for the corresponding predicate $\pi$ is required in $e'$.

In the current setting, predicates are expressions of the form $(r \backslash a)$ whose evidence is the address space ID for the particular $a$. The calculation of evidence is described by the rules in Figure 9, which are direct extensions of the earlier rules for predicate entailment that were given in Figure 4. Intuitively, a derivation of $P \models e : \pi$ tells us that we can use $e$ as evidence for the predicate $\pi$ in any environment where the assumptions in $P$ are valid. The second rule is the most interesting and tells us how to find the address space ID in a row $\{a \mid r\}$:

$$P \cup \{v : \pi\} \models v : \pi$$

$$\frac{P \models e : (r \backslash a)}{P \models m : \{a' \mid r\} \backslash a} \quad m = \begin{cases} e, & a < a' \\ e + 1, & a' < a \end{cases}$$

$$P \models () : (\{\} \backslash a)$$

**Figure 9.** Predicate entailment for rows.

- If $a$ comes before $a'$ in the total ordering, $<$, on address space IDs, then the required ID will be the same as the ID $e$ of $a$ in $r$.
- But, if $a'$ comes before $a$, then we need to use an ID of $e + 1$ to account for the address of $a'$.

In general, these rules calculate address IDs that are either a fixed natural number, or an addition from a natural number and one of the variables in $P$.

For reasons of space, we omit the complete description of translation from this paper, and instead focus on describing the two rules that account for the user and introduction of address space ID parameters. The first of these is a variation on function application:

$$\frac{P|\Gamma \vdash e \leadsto e' : \pi \Rightarrow \rho \quad P \models e'' : \pi}{P|\Gamma \vdash e \leadsto e' \; e'' : \rho}$$

$$\frac{}{\{\}|\{\} \vdash^W ; : \texttt{void}} [\textsc{Empty}] \quad \frac{P|T\Gamma \vdash^W e : \tau \quad Q|T'T\Gamma \vdash^W z : \tau'}{T'P \cup Q|T'T\Gamma \vdash^W \texttt{return}\, e; z : \tau} [\textsc{Ret}] \quad \frac{P|T\Gamma \vdash^W e : \tau' \quad Q|T'T\Gamma \vdash^W z : \tau}{T'P \cup Q|T'T\Gamma \vdash^W e; z : \tau} [\textsc{Expr}]$$

$$\frac{(x : \forall \alpha_i.P \Rightarrow \tau) \in \Gamma \quad \beta_i new}{[\beta_i/\alpha_i]P|\Gamma \vdash^W x : [\beta_i/\alpha_i]\tau} [\textsc{Var}] \quad \frac{P|T\Gamma \vdash^W e : \tau'' \quad T\tau'' \stackrel{U}{\sim} \tau \quad Q|T'T\Gamma, v : \tau \vdash^W z : \tau'}{U(T'P \cup Q)|UT'T\Gamma \vdash^W \tau\, v = e; z : \tau'} [\textsc{VDecl}]$$

$$\frac{P|T\Gamma \vdash^W e : \tau \quad \tau \stackrel{U}{\sim} bool \quad Q|T'T\Gamma \vdash^W z_0; z : \tau' \quad Q'|T''T'T\Gamma \vdash^W z_1; z : \tau'' \quad U\tau' \stackrel{U'}{\sim} U\tau''}{U'U(P \cup Q \cup Q')|U'UT''T'T\Gamma \vdash^W \texttt{if}\, (e)\, \{z_0\}\, \texttt{else}\, \{z_1\}; z : U'U\tau''} [\textsc{If}]$$

$$\frac{P|T\Gamma \vdash^W e : \tau \quad \tau \stackrel{U}{\sim} bool \quad Q|T'T\Gamma \vdash z : \tau' \quad U\tau' \stackrel{U'}{\sim} void \quad Q'|T''T'T\Gamma \vdash z' : \tau''}{U'U(P \cup Q \cup Q')|U'UT''T'T\Gamma \vdash \texttt{while}\, (e)\, \{z\}; z' : \tau''} [\textsc{While}]$$

$$\frac{P|T\Gamma_{v_i}, v_i : \alpha_i \vdash^W b : \tau \quad \alpha_i\ new}{P|T\Gamma \vdash^W \tau\, x(v_i)\, \{b\} : T\alpha_i \to \tau} [\textsc{Fcn}]$$

$$\frac{\begin{array}{c} P|T'\Gamma \vdash^W e : (t_i \to) \quad Q_0|T_oT'\Gamma \vdash^W e_0 : \tau_0 \quad \ldots \quad Q_n|(T_0 \ldots T_{n-1})T'\Gamma \vdash^W e_n : \tau_n \\ t_0 \stackrel{U_0}{\sim} \tau_0 \quad \ldots \quad (U_{n-1}(\ldots(U_0))t_n \stackrel{U_n}{\sim} U_{n-1}(\ldots(U_0))\tau_n \to \alpha \quad \alpha\ new \end{array}}{\bigcup((U_n \ldots U_0)(T_0 \ldots T_nT')P, Q_i)|(U_n \ldots U_0)(T_0 \ldots T_nT')\Gamma \vdash^W x(e_i) : (U_n \ldots U_0)\alpha} [\textsc{App}]$$

$$\frac{P_i|T_i\Gamma \vdash^W f_i(v_i)\{b\} : \tau_i \quad \sigma_i = Gen(T_i\Gamma, P_i \Rightarrow \tau_i) \quad P'|T'(T_0 \ldots T_n\Gamma_{f_i}, f_i : \sigma_i) \vdash^W b' : \tau' \quad \tau' \stackrel{U}{\sim} void}{P'|T'(T_0 \ldots T_n)\Gamma \vdash^W f_i(v_i)\{b_j\}\, \texttt{main}()\{b'\} : U\tau'} [\textsc{Program}]$$

**Figure 8.** Type inference algorithm W.

This tells us that we need to supply suitable evidence $e''$ in the translation of any program whose type is qualified by a predicate $\pi$. The second rule is analogous to function abstraction, and allows us to move constraints from the predicate assignment $P$ into the inferred type[5]:

$$\frac{P \cup \{v : \pi\}|\Gamma \vdash e \rightsquigarrow e' : \rho}{P|\Gamma \vdash e \rightsquigarrow \lambda v.e' : \pi \Rightarrow \rho}$$

These two rules are direct extensions of the $(\Rightarrow E)$ and $\Rightarrow I)$ in Figure 5, and combined with simple extensions of the other rules there, we can construct a translation for any term in MiniAS.

## 6. Down casting

As of today the OpenCL 1.x specification does not allow for casting between pointers of different address spaces, it seems to make little sense when address spaces are disjoint. With the introduction of generic address spaces the ability to up cast to generic address space is built into the type system by default. However, the ability to down cast (i.e. translate from a generic address space to a specialized one) may also useful. Moreover combining down casting with the ability to test if a generic address pointer is in a given specialized address space allows one to call specialized library functions.

[5] For simplicity we have "cheated" a little by introducing the use of lambda abstraction not defined in MiniAS. However, in practice evidence abstraction will only appear at function application and so an implementation would just add the additional argument to the associated function and thus there is no additional overhead or complexity.

For example, consider the case when a $3^{rd}$ party library contains the specialized functions:

```
int foo_local(local * int);
int foo_global(global * int);
```

but does not contain a generic version. Using a cast operator similar to C++'s `dynamic_cast`—that either casts a generic address space to the specified address space, if it indeed matches the actual address space, or returns NULL—it is straightforward to define a generic version of $foo$:

```
int foo(generic * int p)
{
    if ((global int * gptr =
         dynamic_cast<global int *>(p)) != NULL) {
           return foo_global(gptr);
    }
    else if ((local int * lptr =
         dynamic_cast<local int *>(p)) != NULL) {
            return foo_local(lptr);
    } else {
        return -1;
    }
}
```

`dynamic_cast<_>(_)` can be assigned the type:

$$\forall \alpha, r.r \backslash a \Rightarrow\ ASpace\, \{a|r\} * \alpha \to a * \alpha$$

with specific versions implemented for each value of $a$, i.e. *global*, *local*, and *private*. For example after translation pseudo code for the *global* address space version might be defined as:

```
dynamic_cast_global(idx, ptr)
{
   if (idx == 0) {
       return ptr;
   } else {
       return NULL;
   }
}
```

Of course, a compiler targeting specific hardware that supports generic address spaces and instruction set that defines the ability to convert two and from specialized address spaces could generate code directly to these operations. For example, AMD's Heterogeneous System Architecture (HSA) [1, 26] supports a selection of memory segments, most of which are disjoint (similar to OpenCL) and a *flat* address space that subsumes all others. HSA's Input Language (HSAIL) is a low-level device independent ISA supporting the following operations to translate to and from generic (*flat*) address spaces (*segments*):

- Test if a *flat* pointer is in segment:

  ```
  segmentp_segment_b1 dst, src
  ```

- Convert *flat* pointer to segment:

  ```
  ftos_segment_type dst, src
  ```

- Convert segment pointer to *flat*:

  ```
  stof_segment_type dst, src
  ```

Assuming a set of compiler builtin functions mapping directly to HSAIL operations, then $dynamic_cast$ might be implemented directly as:

```
dynamic_cast_global(ptr)
{
   if (segmentp_global(ptr)) {
       return ftos_segment_i32(ptr);
   } else {
       return NULL;
   }
}
```

As HSAIL directly supports a notion of generic address space, then as discussed in Section 2 the address space ID argument has been elided by the compiler with no additional performance cost, even in the presence of dynamic typing.

## 7. Related work

To our knowledge we are the first to propose formalizing OpenCL's address spaces and provide a complete type inference system. Of course, there has been other approaches to abstracting user managed memories and also in the area of type inference for type qualifiers. In this section we discuss work most relevant to ours.

### 7.1 A theory of qualifiers

We are not the first to study type inference for type qualifiers for C and the most relevant is the work of Foster et al [5]. They describe a system that is capable of inferring static type qualifiers, including the ability to have polymorphic functions parameterized by type qualifiers. However, their system is limited to type qualifiers that can be uniquely determined at compile time. They provide no system for dynamically selecting between different types of qualifiers as is necessary in the general case and in particular for address space load and store operations.

Forster et al system has the goal to enforce and discover static invariants that can help the compiler produce more efficient code and rule out unintended program errors. Their system does provide the ability to refine a types qualification, i.e. add const to a pointer, but lacks the capability for the same value to have multiple alternatives for a particular qualifier. In fact their sub-type relation is closely related to Gaster and Jones' [9] extensible record type which introduces a similar lattice type structure.

Additionally Foster et al's system is based on a complicated system of sub-typing constraints which can often lead to complicated types which are difficult for developer to write down in practice. Our system, on the other hand, has simple types which can be easily written down by the developer, although it is not required.

One area of future work would be to incorporate a variant of Gaster and Jones' extensible records into our system to handle the qualifiers of Foster et al. In particular, as such as system would support our compilation scheme, then they might be interesting qualifiers not expressible in Foster et al's system.

### 7.2 OpenCL C++

OpenCL C++ [8] supports OpenCL C address spaces but comes with the additional complication of a classes *this*. The *this* pointer complicates matters as it is often left implicit but effects how particular member functions will behave. For example, consider a classes copy constructor which may be implicitly generated by the compiler from a particular use case. In general, the copy constructor is of the form:

```
Foo(Foo & rhs);
```

But for OpenCL C++, `Foo&` must live in an address space, which if implicitly defined, like the copy constructor itself, must be inferred at compile time. For this OpenCL C++ uses a subset of the type inference algorithm formalized in this paper, whose address spaces are all uniquely inferred at compile time. In developing OpenCL C++ AMD initially implemented an ad-hoc type inference algorithm for address spaces, which time and time again caused pain in the discovery of corner cases that had not been originally considered. This was only conflated with the move to C++11 [11], which has its own inference rules for *auto* and *decltype*. Motivated by these short comings in the original implementation of OpenCL C++ and a desire to add generic address spaces to OpenCL C we were motivated to develop the system described in this paper.

### 7.3 Heterogeneous Parallel Pattern

The GPGPU programming model Heterogeneous Parallel Patterns (HPP) [7] is a braided parallel model, supporting both task and data-parallelism, that is embedded into C++11. Like OpenCL C++ it supports a device programming language that is designed to target GPUs and other accelerator like devices. However, unlike OpenCL C++ it does not expose explicitly managed address spaces and instead it supports a PGAS style globally shared memory model combined with a hierarchical array abstraction, called *DistArray*. Like address spaces *DistArray* is intended to allow developers to explicitly manage data locality, however, this is achieved via describing a hierarchical nesting of regions bound on use.

A key difference when compared to our system is that data placement for HPP's *DistArray* is dynamically determined by the runtime, while our system is based on static type inference. Of course, in the case that an address space cannot be uniquely determined at compile time our system still requires a runtime parameter to determine which memory to read and write from. In developing our system we were constrained to design a system that fitted with an existing programming language (OpenCL), and did not have the freedom to design a language without address spaces.

## 8. Conclusion

We have described a flexible type system for generic (polymorphic) address spaces with effective type inference algorithm and compilation method. A prototype implementation has been written as a standard alone compilation flow allowing developers an researchers alike to study the algorithm in isolation from a more complicated system, such as OpenCL. An implementation of our algorithm has also been implemented as part of AMD's APP SDK, which supports a variant of OpenCL C++. Our experience to date shows that these implementations work well in practice. Furthermore, generic address spaces have been proposed as a future feature of standard OpenCL C and the algorithm proposed in this paper can form a foundation for this development [23].

More generally our system can form the basis of an address space system for an variant of Embedded C and to our knowledge we are the first to propose such as system. Such a system could be used as the foundation to extend Embedded C to C++, similar to OpenCL C++'s extension to OpenCL C, and could prove useful for future embedded system development. Without similar type inference system for address spaces it is hard to see how Embedded C++ could be useful in practice.

The above extensions to OpenCL are, at present, restricted to compile time specialization, i.e. all predicates must be discharged statically otherwise it's a compile time error. Our system is more general and provides the ability to support a dynamic generic address space, via the introduction of address space IDs that are passed around at runtime and used to dynamically select the specific address space operation. While this could have a potential runtime impact we highlight this is only observed when dynamic features are used by the developer. Moreover, we demonstrated that if the underlying system directly supports generic address spaces, as per AMD's HSAIL or Nvidia's Fermi, then the additional address space IDs can be elided, without any additional runtime overhead.

One interesting area of future work is to consider providing a formal type system for something like HSAIL, including handling of its segments and *flat* address space. The system proposed in this paper may in turn be an interesting place to begin that work. For one it describes the semantics of address spaces (i.e. segments) but moreover Jones has shown that more generally the theory of qualified types can be used to statically type and verify intermediate languages such as Java's bytecode [15].

## 9. Acknowledgements

The authors would like to thank Mike Chu of AMD and Garret Morris of Portland State University for their feedback during the development of this work.

## References

[1] Advanced Micro Devices (AMD). HSA Programmer's Reference Manual. http://developer.amd.com, 2012.

[2] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Inf. Comput.*, 93(1):172–221, July 1991.

[3] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 273–280, New York, NY, USA, 1989. ACM.

[4] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[5] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *Proceedings of the ACM conference on Programming language design and implementation*, PLDI '99, pages 192–203, New York, NY, USA, 1999. ACM.

[6] B. R. Gaster. *Records, variants, and qualified types*. PhD thesis, University of Nottingham, August 1998.

[7] B. R. Gaster and L. Howes. Can GPGPU programming be liberated from the data-parallel bottleneck? *IEEE Computer*, August 2012.

[8] B. R. Gaster and L. Howes. OpenCL C++. In *Sixth Workshop on General Purpose Processing Using GPUs (GPGPU 6)*, 2013.

[9] B. R. Gaster and M. P. Jones. A Polymorphic Type System for Extensible Records and Variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, Nov. 1996.

[10] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc*, 146:29–60, December 1969.

[11] ISO/IEC. Progrmming languages C++. ISO/IEC 14882:2011(E), 2011.

[12] ISO/IEC. Progrmming languages Embedded C. ISO/IEC DTR 18037, 2011.

[13] M. P. Jones. A theory of qualified types. In *Symposium proceedings on 4th European symposium on programming*, ESOP'92, pages 287–306, London, UK, UK, 1992. Springer-Verlag.

[14] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 52–61, New York, NY, USA, 1993. ACM.

[15] M. P. Jones. The functions of java bytecode. In *In Workshop on the Formal Underpinnings of the Java Paradigm*, 1998.

[16] S. P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, September 2002.

[17] M. Mantor and M. Houston. AMD Graphics Core Next: Low power high performance graphics and parallel compute. In *High Performance Graphics Conference*, Hot3D, 2011.

[18] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[19] J. G. Morris and M. P. Jones. Instance chains: type class programming without overlapping instances. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 375–386, New York, NY, USA, 2010. ACM.

[20] Nvidia. NVIDIAs Next Generation Cuda Compute Architecture: Fermi. *Whitepaper*, 2010.

[21] NVIDIA Corporation. NVIDIA CUDA programming guide, version 4.2, 2012.

[22] OpenCL Working Group. The OpenCL specification, version 1.2, revision 16. Khronos, 2011.

[23] OpenCL Working Group. private communication, 2012.

[24] D. Rémy. Type inference for records in a natural extension of ml. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, 1994.

[25] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, Jan. 1965.

[26] N. Rubin and B. R. Gaster. An overview of HSAIL. In *AMD Fusion developer summit*, 2012.