One benefit of the invention of programming languages is that the same program can be compiled to and run on multiple platforms without having to be rewritten. If a compiler writer constructs a back-end for a new platform all the programs written in the language can be recompiled and reused on it. In the best case this can be done without having to change the source code at all. The burden of handling the platform-specific details is no longer on the application programmer.

The situation changes if we broaden the expectations of our compilers. It is not the case that we can take a program written for a “conventional” computer architecture and recompile it to run on a distributed computer cluster. Even going from one system configuration to another, e.g. adding a node to the system, can require substantial changes to the program. With few exceptions, we have to rewrite our programs using domain-specific languages or libraries. Suddenly it is again the application programmer’s burden to take care of system-specific details.

In [3], we present the theoretical basis for a compiler for ICA [6], a higher-order programming language with shared state concurrency, targeting distributed systems. The current presentation discusses the implementation of this compiler.

The approach that this project takes is inspired by ideas from programming language semantics, and relies on the idea that computation can be decomposed into communication or interaction. This idea has previously been explored in several different forms, including the Geometry of Interaction [7], the translation of \(\lambda\)-calculus into \(\pi\)-calculus [11], and game semantics [9]. It is of special significance that these examples are not only of denotational nature, but also operational in the sense that they can form a basis for abstract machines and compilation (see [10, 12, 2, 1, 5]).

In [3], we construct Heap and Register Abstract Machine (HRAM) nets, which model conventional, multi-threaded computers with a small instruction set for heap manipulation that communicate over an asynchronous point-to-point network. Taking the denotation of an HRAM net to be the set of communication traces that it permits, we can compare it to the denotation of ICA programs,
in a formulation of game semantics in the nominal model \cite{4}, and construct HRAMs that implement (behave exactly like, as long as they get valid inputs) the corresponding strategies. We thus use game semantics as a specification for HRAM nets.

The benefit of the nominal model compared to other models is that pointer manipulation does not require any encoding or decoding, as in integer-based representations, but exploits the ability of HRAMs to create locally fresh heap pointers. Second, the message token size is constant, as only names are passed around; the computational history of a token is stored by the HRAMs rather than passing it around (cf. GoI, IAM).

These abstract machines are sufficiently low-level to be used as a basis for compilation, and in this presentation we show how they can be compiled to C, using MPI \cite{8} for communication. The compiler enables seamless computing by letting the programmer specify the network location of parts of the program, and then automatically handling the communication required.

The current implementation uses pragma-like code annotations to specify the location of part of a program, e.g.:

\[
\{\text{new } x. \ \text{x := } \{f(x)\}@B + \{g(x)\}@C\}@A
\]

Here \(x\) is a variable located on the node named \(A\), that functions \(f\) and \(g\) located on nodes \(B\) and \(C\) (the innermost node annotation is used), have read-write access to. We see that it is possible to access non-local variables, and in general the programmer is free to add a location annotation to any sub-term in the program, allowing e.g. higher-order functions across node boundaries.

We present some of the peculiarities of using this interaction-based model:

**Garbage collection** The combination of using explicit diagonals and a call-by-name evaluation strategy obviates the need for garbage collection, which is desirable in a distributed system. However, it also incurs an overhead at runtime. We show that we can introduce syntactic sugar for call-by-value function application at base type by using local state, and construct a special-purpose HRAM for application, which combines the functionality of the diagonal and composition.

**Efficiency** A naïve implementation is flexible but costly when running programs locally. We discuss the possibility of compiling the local parts of a program using conventional techniques, and only using the game machinery at the node boundaries, taking inspiration from \cite{13}.

With these developments in place, we hope to take another step towards enabling seamless distributed computing.

\footnote{Source code available at \url{http://veritygos.org/gams}}
References


