High-Performance Embedded Systems-on-a-Chip

Lecture 12: Executing Alpha

Sanjay Rajopadhye

Computer Science, Colorado State University
Operational Semantics
Operational Semantics

Scanning polyhedra
Outline

- Operational Semantics
- Scanning polyhedra
- Horrendously inefficient code generation
Outline

- Operational Semantics
- Scanning polyhedra
- Horrendously inefficient code generation
- (In)efficient code: allocation memory for domains
Outline

- Operational Semantics
- Scanning polyhedra
- Horrendously inefficient code generation
- (In)efficient code: allocation memory for domains
- Efficient code generation
Operational Semantics of Alpha (expressions)

- Expressions denote mappings from indices to values
- Define a function $\text{Eval} : \langle\text{Exp}\rangle \times \mathbb{Z}^n \rightarrow \text{Type}$ that actually (operationally) computes this mapping.

$$\text{Eval}(\langle\text{exp}\rangle, z) = \begin{cases} 
\text{Eval}'(z) & \text{if } z \in \mathcal{D}(\text{exp}) \\
\bot & \text{otherwise}
\end{cases}$$

- Eval is defined recursively
- Six syntax rules $\Rightarrow$ six cases
The Eval’ function

- \text{Eval}'(\langle \text{Const} \rangle, z) = C'
The Eval' function

- $\text{Eval}'(\langle \text{Const} \rangle, z) = C'$
- $\text{Eval}'(\langle E_1 \rangle \text{ op } \langle E_2 \rangle, z) = \text{Eval}'(\langle E_1 \rangle, z) \oplus \text{Eval}'(\langle E_2 \rangle, z)$
The Eval' function

- \( \text{Eval}'(\langle \text{Const} \rangle, z) = C' \)
- \( \text{Eval}'(\langle E_1 \rangle \text{op} \langle E_2 \rangle, z) = \text{Eval}'(\langle E_1 \rangle, z) \oplus \text{Eval}'(\langle E_2 \rangle, z) \)
- \( \text{Eval}'(\langle \text{case..} E_i \langle \text{..esac, } z \rangle = \)

\[
\begin{align*}
&\vdots \\
&\text{Eval}'(\langle E_i \rangle, z) \text{ if } z \in D(\langle E_i \rangle) \\
&\vdots
\end{align*}
\]
The \textit{Eval'} function

- \(\text{Eval}'(\langle\text{Const}\rangle, z) = C'\)
- \(\text{Eval}'(\langle E_1 \rangle \text{op} \langle E_2 \rangle, z) = \text{Eval}'(\langle E_1 \rangle, z) \oplus \text{Eval}'(\langle E_2 \rangle, z)\)
- \(\text{Eval}'(\text{case..} \langle E_i \rangle \text{..esac, } z) = \)
  
  \[
  \begin{cases}
  \vdots \\
  \text{Eval}'(\langle E_i \rangle, z) & \text{if } z \in \mathcal{D}(\langle E_i \rangle) \\
  \vdots 
  \end{cases}
  \]
- \(\text{Eval}'(D : \langle E \rangle, z) = \text{Eval}'(\langle E \rangle, z)\)
The Eval’ function

- \( \text{Eval}'(\langle \text{Const} \rangle, z) = C \)
- \( \text{Eval}'(\langle E1 \rangle \text{op} \langle E2 \rangle, z) = \text{Eval}'(\langle E1 \rangle, z) \oplus \text{Eval}'(\langle E2 \rangle, z) \)
- \( \text{Eval}'(\text{case}..\langle Ei \rangle..\text{esac}, z) = \begin{cases} 
\vdots \\
\text{Eval}'(\langle Ei \rangle, z) \text{ if } z \in D(\langle Ei \rangle) \\
\vdots 
\end{cases} \)
- \( \text{Eval}'(D : \langle E \rangle, z) = \text{Eval}'(\langle E \rangle, z) \)
- \( \text{Eval}'(\langle E \rangle.f, z) = \text{Eval}'(\langle E \rangle, f(z)) \)
The Eval' function

- \( \text{Eval}'(\langle \text{Const} \rangle, z) = C' \)
- \( \text{Eval}'(\langle E_1 \rangle \oplus \langle E_2 \rangle, z) = \text{Eval}'(\langle E_1 \rangle, z) \oplus \text{Eval}'(\langle E_2 \rangle, z) \)
- \( \text{Eval}'(\text{case..} \langle E_i \rangle \ldots \text{esac}, z) = \begin{cases} \vdots & \text{if } z \in D(\langle E_i \rangle) \end{cases} \)
- \( \text{Eval}'(D : \langle E \rangle, z) = \text{Eval}'(\langle E \rangle, z) \)
- \( \text{Eval}'(\langle E \rangle.f, z) = \text{Eval}'(\langle E \rangle, f(z)) \)
- \( \text{Eval}'(\langle \text{Var} \rangle, z) = \text{EvalVar}(z) \)
Semantics of Equations

- Equations do **not denote** mappings from indices to values
Semantics of Equations

- Equations do not denote mappings from indices to values
- Denotational Semantics: Equations denote “additions” to a “store of definitions”
Semantics of Equations

- Equations do not denote mappings from indices to values

- Denotational Semantics: Equations denote “additions” to a “store of definitions”

- Operational Semantics:
Semantics of Equations

- Equations do **not denote** mappings from indices to values
- **Denotational Semantics:** Equations denote “additions” to a “store of definitions”
- **Operational Semantics:**

\[
\text{Eval}(\text{var} = \langle \text{Exp} \rangle) = (\text{defun EvalVar(Eval(\langle Exp \rangle))})
\]
Semantics of programs (systems)

- Denotational Semantics: programs denote mappings from input variables to output variables
- (strict) Operational Semantics:
  1. Read Input Variables
  2. Compute (Local) and Output Variables
  3. Write Output Variables
Scanning Polyhedra

- Given a polyhedron \( \mathcal{P} \)
- Problem: (generate code to) visit (in lexicographic order) all the integer points in \( \mathcal{P} \)
Nonstrict Operational Semantics

1. Read Input Variables
2. Write Output Variables (computing only those local variables that are necessary)

this will yield horribly inefficient code
Nonstrict Operational Semantics

1. Read Input Variables
2. Write Output Variables (computing only those local variables that are necessary)

this will yield horribly inefficient code
Improvement

- Use **memoization** to avoid recomputation
- Allocate memory, store previously computed values, and evaluate only (at most) once
Main Drawback: Too many context switches – factor of 5 to 8 for simple examples
**Main Drawback:** Too many context switches – factor of 5 to 8 for simple examples

**Solution:** Determine a schedule, and visit the points in the domains of the variables in that order.
**Main Drawback:** Too many context switches – factor of 5 to 8 for simple examples

**Solution:** Determine a schedule, and visit the points in the domains of the variables in that order

**Key issues:**
- How to determine a schedule
- How to exploit this to generate code (scanning unions of polyhedra)