Designing intermediate representations

CS448h
Oct. 6, 2015
Programming languages are all about representations of computation.

The right representations are what give DSLs their power.

DSLs are often best designed from the IRs out.
For example: linear algebra \[ x' = ABx \]

\[ A : L \times M \]
\[ B : M \times N \]
\[ x : N \times 1 \]
For example: linear algebra  \[ x' = ABx \]

\[ A : L \times M \]
\[ B : M \times N \]
\[ x : N \times 1 \]

\[ C : L \times N \]
for \( l \) in \( L \):
  for \( m \) in \( M \):
    for \( n \) in \( N \):
      \[ C[l,n] \leftarrow A[l,m] \times B[m,n] \]
for \( l \) in \( L \):
  for \( n \) in \( N \):
    \[ x'[l] \leftarrow C[l,n] \times x[n] \]
For example: linear algebra  \( x' = ABx \)

\[ \begin{align*}
C & : L \times N \\
\text{for } l \text{ in } L : \\
& \quad \text{for } m \text{ in } M : \\
& \quad \quad \text{for } n \text{ in } N : \\
& \quad \quad \quad C[l,n] \leftarrow A[l,m] \times B[m,n] \\
\text{for } l \text{ in } L : \\
& \quad \text{for } n \text{ in } N : \\
& \quad \quad x'[l] \leftarrow C[l,n] \times x[n]
\end{align*} \]

\[ x' = \text{mul}( \text{mul}( A, B ), x ) \]
For example: linear algebra

\[ x' = ABx \]

\[ A : L \times M \]
\[ B : M \times N \]
\[ x : N \times 1 \]

\( C : L \times N \)

for \( l \) in \( L \):
  for \( m \) in \( M \):
    for \( n \) in \( N \):
      \( C[l,n] += A[l,m] \times B[m,n] \)

for \( l \) in \( L \):
  for \( n \) in \( N \):
    \( x'[l] += C[l,n] \times x[n] \)

\( x' = \text{mul}( \text{mul}( A, B ), x ) \)
For example: linear algebra

\[ x' = ABx \]

\[ A : L \times M \]
\[ B : M \times N \]
\[ x : N \times 1 \]

\[ C : L \times N \]
for \( l \) in \( L \):
  for \( m \) in \( M \):
    for \( n \) in \( N \):
      \[ C[l,n] += A[l,m] \times B[m,n] \]

for \( l \) in \( L \):
  for \( n \) in \( N \):
    \[ x'[l] += C[l,n] \times x[n] \]

\[ x' = \text{mul}( \text{mul}( A, B ), x ) \]

???

simple rewrite

\[ x' = \text{mul}( A, \text{mul}( B, x ) ) \]
What makes a good IR?

simplicity
  as few types as possible

generality / expressive power

analyzability / transformability
  restriction
Different representations are best for different problems.

across domains

why we make DSLs!

for different compilation problems in a single domain

not 1 IR per compiler/DSL, but many!
What makes a good IR? (take 2)

Easy *target* to generate from what came before

Easy *source* from which to generate what comes after
What makes a good IR? (take 2)

Easy *target* to generate from what came before

Easy *source* from which to generate what comes after

*at the front-end: easy for a human to write!*
Common types of representation

trees reflect the hierarchical structure of programs

graphs reflect control and data flow
Common types of representation

trees reflect the hierarchical structure of programs

graphs reflect control and data flow

tables map identifiers to nodes, auxiliary metadata
Common types of representation

AST: user code

High-level: user intent

Low-level: execution strategy

Instruction-level: machine operations
Common types of representation

AST: *user code*

High-level: *user intent*

Low-level: *execution strategy*

Instruction-level: *machine operations*

(lowering)
Algebraic Data Types

a notation for representations
Algebraic Data Types

a notation for representations

\[ x = A \mid B(y) \mid C(x, y) \]
\[ y = D(x) \]
Algebraic Data Types

a notation for representations

\[ x = A \mid B(y) \mid C(x, y) \quad B(D(C(A, D(A)))) \]
\[ y = D(x) \]
Algebraic Data Types
a notation for representations

\[ x = A \mid B(y) \mid C(x, y) \]
\[ y = D(x) \]

\[ \text{list} = \text{Cons}(\text{val}, \text{list}) \mid \text{Nil} \]
Algebraic Data Types

a notation for representations

\[
x = A \mid B(y) \mid C(x, y)
\]

\[
y = D(x)
\]

\[
\text{list} = \text{Cons}(val, \text{list}) \mid \text{Nil}
\]

\[
\text{list} = \text{Cons}(val, \text{list}) \mid \text{Atom}(val)
\]
Representing Regexs & NFAs
Representing Regexes & NFAs

\[ re = \text{Char} \ ( \text{char} ) \]
\[ \text{| Seq} \ ( \text{re list} ) \]
\[ \text{| Or} \ ( \text{re list} ) \]
\[ \text{| Star} \ ( \text{re} ) \]
\[ \text{| Maybe} \ ( \text{re} ) \]
Representing Regexs & NFAs

\[ \text{nfa} = \text{NFA} \left( \text{node list, start : node} \right) \]
Representing Regexes & NFAs

\[ \text{nfa} = \text{NFA} ( \text{node list, start : node} ) \]

\[ \text{node} = \text{Node} ( \text{edge list, accepts : bool, id : int} ) \]
Representing Regexs & NFAs

nfa = NFA ( node list, start : node )

node = Node ( edge list, accepts : bool, id : int )

edge = EpsEdge ( pointsTo : int )
  | CharEdge ( token : char, pointsTo : int )

re = Char ( char )
  | Seq ( re list )
  | Or ( re list )
  | Star ( re )
  | Maybe ( re )
Representing Regexs & NFAs

\[ nfa = NFA \ ( \text{node list, start : node} ) \]

\[ \text{node} = \text{Node} \ ( \text{edge list, accepts : bool, id : int} ) \]

\[ \text{edge} = \text{EpsEdge} \ ( \text{pointsTo : int} ) \]
  \[ \text{CharEdge} \ ( \text{token : char, pointsTo : int} ) \]

\[ \text{nodemap} = \text{map int} \rightarrow \text{node} \]
Why is this a good idea?

IRs are naturally recursive data structures with variants

Concise notation to formalize what we’re building

Writing down early reveals issues
Common ways to fail

Throw away information

*including what’s in the code vs. the programmer’s head*

Be too general

Turing completeness is a curse

*when in doubt, restrict rather than generalize!*

*Expect to get your IRs wrong at first!*
Design from your representations out!
Iterate until they feel right