Bluespec Scheduling

Joe Stoy

17 January, 2003

Times and Rules

17 January, 2003
O Tempora, O Mores

sandburst

17 January, 2003

What I Did on My Holidays

sandburst

17 January, 2003
What I Did on My Holidays

• We had a working project;
• Mieszko and I changed the way the compiler prioritized rules;
• The project spectacularly stopped working;
• We changed it back.

MORAL: These decisions are far too important to be left to the compiler.

Agenda

• Why did it go wrong?
• What can we do about it?
  – Model (what are we trying to do?)
  – Present situation
  – Notation
  – Proposals
Why Did It Go Wrong?

- Previously the compiler prioritized (roughly) in order of the rules’ appearance in the text.
- We changed it to prioritize in alphabetical order of the rules’ labels.

Programming Errors

- Foo Bar effect
  - Natural order different from alphabetical order

```
"uc random":
  when True ==> action
  x <- randNo.get
  ucRand.put x

"mc random":
  when True ==> action
  x <- randNo.get
  mcRand.put x
```
Programming Errors

• Foo Bar effect
• Catch-all case
  – Several rules decrement a counter and take various actions, depending on its value
  – The default case merely decrements the counter
  – Ought to have been mutually exclusive (e.g. by “<+” operator), but I forgot
  – OK at bottom of list, but dominated some of the others when in alphabetical order
  – But saying it properly is syntactically unpleasant.

Programming Errors

• Foo Bar effect
• Catch-all case
• Add the Library, and stir
  – Modules usually imported at head of program
  – Their rules clump together, usually with higher priority than rules using imported modules’ methods
  – Acceptable, most of the time
  – In alphabetical order, all hell breaks loose!
    • No uniform naming scheme works
    • Completely breaks modularity
Programming Errors

• Foo Bar effect
• Catch-all case
• Add the Library, and stir

• Foo Bar effect
  – Need to specify priority
• Catch-all case
  – Need convenient way of specifying pre-emption, exclusivity etc.
• Add the Library, and stir
  – Need to handle interaction of rules and methods, in a way which respects modularity.
Model of What We’re Trying To Do

- Two inter-related tasks, for each particular execution:
  - **Sequencing:** choosing the sequence of rules for firing, one at a time, according to TRS
  - **Chunking:** partitioning this sequence into chunks, each of which fires on a single clock.
    - Cf. operation of a sausage machine. The scheduler gets to go between each chunk.
    - The first is important for partial correctness; the second deals more with performance issues.

Aims

- Whenever compiler makes an “interesting” choice, it should utter a warning.
- This should include a fragment of text, which the programmer may add to the program, to specify how the choice should be resolved.
- The programmer may edit the fragment before incorporation if appropriate.
- (Might also dump and restore whole schedules, but that’s different.)
The Present Scheduler-Generator

- Divides rules up into “cliques”, each of which may be scheduled completely independently of the others.
- One-rule cliques, and members of cliques of mutually exclusive rules, can fire whenever enabled, with their own (trivial) schedulers.
- The degenerate case, but common and important.
- Two other kinds of scheduler.

(1) Direct Scheduler

- Consider each subset of rules in a clique.
- Assuming just that subset is enabled, select which of its elements should fire.
- Selection done greedily, according to some notion of priority.
- Good, but O(2^n).
(2) Priority Group Scheduler

- Used when direct schedule generation too slow.
- Consider each rule in clique.
- Assuming that’s the highest priority rule enabled, form set from remaining rules such that each element may fire if enabled.
- Selection of elements again greedy.
- \(O(n^2)\), but less good.
- Bad case:
  - Rules \(a, b, c\) (in that order of priority)
  - \(b\) or \(c\) can fire with \(a\), but not both.
  - \(c\) always overlooked, even if \(b\) not enabled.
- (But improvement on old PriPar method.)

Agenda

- Why did it go wrong?
- What can we do about it?
  - Model (what are we trying to do?)
  - Present situation
  - Notation
  - Proposals
Notation

- Several kinds of annotation
- Assertions (claims)
  - “I believe this is true: please verify or disprove.”
  - Doesn’t affect semantics of program
  - May conveniently be a pragma

\{ -# ASSERT fire when enabled #-\}

- Prescriptions (fiats)
  - “This is extra information: please use it when appropriate.”
  - Does affect semantics; shouldn’t be a pragma.

priority "a" "b"

Assumptions

- “This information is true, but you might not be smart enough to work it out for yourself: please use it (but give an error if you can prove me wrong).”

\begin{itemize}
  \item rules
    \begin{itemize}
    \item "a": when x==0 ==> action1
    \item "b": when y==0 ==> action2
    \end{itemize}
  \end{itemize}

-- INARIANT: x==0 implies y/=0

assume exclusive "a" "b"

- Insistence (Overruling)
  - “You may safely assume this to be true (even though we both know it isn’t): ride roughshod over any indications to the contrary.”

insist "a" \(\langle\rangle\) "b"
Notation

• Kinds of annotation:
  – Assertions
  – Prescriptions
  – Assumptions
  – Overrulings

Notational Details

• Notations apply to (smallest) containing module
• Allow methods to be labelled (like rules)
  – Refer to each by label
  • property a b { c d} e
• Semantics: The property applies to each element, or to transitive pairs: thus in the latter case “property a b c” is equivalent to
  property a b
  property a c
  property b c
• If S is a set such as { c d} , “property a S b” is equivalent to
  ∀ x ∈ S: property a x b
• If m is a method label, it denotes the set of all rules using that method.
Notation: Syntactic trivia

- Separators: , ; or space?
  - Does offside rule apply?
- Rules identified by labels?
- Allow initial prefixes of labels?
  - If prefix identifies several rules, treat them as a set
  - If only one, leads to conciseness in program
  - Longer labels still helpful on waves.
- Allow quotes to be omitted if nude string satisfies variable-identifier syntax?

Properties

- **exclusive** x y
  - “Make x and y mutually exclusive, giving preference to x.”
- This is about enabling, and is dealt with before the scheduler gets to go.
- Thus
  - **exclusive** S1 S2
  - where S1, S2 are set of rules, achieves the same effect as the present “<+”.
- Quite different from **assume exclusive**.
- May also have syntactic sugar within a single **rules** expression:
  - otherwise => defaultAction
Properties

- **urgency** \( x \ y \)
  - “Never omit \( x \) from a chunk for the sake of including \( y \).”
  - If they can both be included, fine!
- Scheduler often considers rules in order of urgency (when doing greedy selection).
- This property is about chunking.
- Says nothing about sequencing order.

Properties

- **preempts** \( x \ y \)
  - “Never allow \( x \) and \( y \) to be in the same chunk, and give preference to \( x \).”
- This is about chunking.
- Note: \( y \) is excluded only if \( x \) actually fires, unlike exclusive (and \( <+ \)).
- Maybe allow
  - \{ -# ASSERT fire unless preempted #-\}
Properties

• **insist** $x < y$  **insist** $x <> y$  etc
  - “I insist that if they’re both enabled, $x$ and $y$ can be composed this way round in the same chunk, overruling any contrary deductions by you, O compiler.”
  - Does not overrule deductions involving a third rule.
• The MIT gang want something like this, for getting round problems in a top-down way.
• The Sandburst gang has hitherto proceeded bottom-up, by using primitives with relaxed constraints.
• (More on this later, if there’s time.)

Properties

• **precedes** $x ~ y$
  - “If they’re both enabled, they must be sequenced in this order.”
• This property is about sequencing (though it may affect chunking).
• If $x$ and $y$ can never be composed this way round, it’s a compile-time error (or just possibly equivalent to pre-empting).
• Maybe not very useful, except for avoiding consequences of naughty insistences.
**Possible Additional Properties**

- **assume exclusive** $a b$
  - Already described

- **direct** $x y$
  - "All cliques containing $x$ or $y$ should be given direct schedules."
  - Note that this may take a very long time.

- **separate** $x$
  - What's a better word for this?
  - "If $x$ is part of a PriGroup, consider ‘$x$ enabled’ and ‘$x$ not enabled’ separately."
  - May double number of cases, but better than direct.

**Separate Compilation**

- These suggestions have been assuming a single compilation.
- A method of a separately compiled modules is treated as a single internal rule.
- Have not yet thought through how these suggestions apply then (though probably OK).
The End
– (except for digression)

Top Down vs. Bottom Up

- Example: a FIFO written in Bluespec
- Goal: allow simultaneous enq and deq when not full and not empty
- Problem:
  "enq_method":
  enq x = increment (asReg i)
  when notFull i j
  "deq_method":
  deq = increment (asReg j)
  when notEmpty i j
- Not sequentially composable!
Top Down vs. Bottom Up

- MIT solution:
  insist “deq_method” <> “enq_method”
  - Top-down

- Sandburst solution:
  use ConfigReg instead of Reg for i and j
  - Allow _read <> _write
  - Bottom-up

MIT’s solution better because...

- Ban on _write<>_read overruled only when necessary, not relaxed everywhere.
Sandburst’s solution better because…

- MIT’s way might not respect TRS semantics
  - E.g. there might be an impossible intermediate state (actually OK in this example)
- Can explain Sandburst’s way entirely in TRS:

```plaintext
mkConfigReg :: a -> Reg a
mkConfigReg x =
  module
  r :: Reg a = mkReg x
  w :: Reg a = mkReg x
rules
  "configReg":
    when True ==> r := w
    -- this fires with bounded delay after each _write
interface
  _read = r
  _write y = w := y
```