Action Transformation:

An Application of Sort Inference

by Kent D. Lee

University of Iowa

Advisor: Ken Slonneger

Overview

- Introduction to Action Semantics
- Automated Compiler Generation
- Example Calculator Language
- Action Transformations

Action Semantics

- Created by Peter Mosses with collaboration of David Watt

- Defines the action as the basic unit of computation

What's the Purpose of an action?

- Actions manipulate three structures
  - Transients - Represented as tuples of data
  - Bindings - Represented as a map of identifiers to bindable data
  - Storage - Represented as a map of cells to storable data
- Actions compute new transients, bindings, and storage from existing transients, bindings, and storage.
  - Incoming transients, bindings, and storage
  - Outgoing transients, bindings, and storage

Example Actions

- allocate a cell
  - sets aside a new cell in storage and gives it as a transient
- bind “M” to 5
  - produces the new binding in the outgoing bindings
- store 6 in the given cell
  - stores 6 by mapping the given cell to the value 6.

Yielders

- Sometimes we want to access the incoming transients, bindings, or storage.

- bind "M" to the **given cell**

  - **given** s is a yielder that accesses the first datum in the incoming transients and insists that it is a subsort of sort s

- **given** $s\#n$ accesses the n^{th} datum of the incoming transients

- s **bound to** id accesses the datum that id maps to in the incoming bindings

- s **stored in** c accesses the datum mapped to by the cell c

Data

- Action Semantics uses a sort system instead of a type system.

- Sorts of data are partially ordered by the subsort relation ($\leq$)

- datum is a sort that contains all individuals

- Other sorts include integers, truth-values, reals, 4, 5, true|false (i.e. truth-value), integer&truth-value, nothing, and data

- $s_1|s_2$ is the sort of the join of s_1 and s_2

- $s_1\&s_2$ is the sort of the meet of s_1 and s_2

Combinators

- Simple Actions can be combined to create complex or compound actions

- Combinators serve two purposes
  - They are the glue for tying actions together
  - They dictate how the incoming and outgoing transients, bindings, and storage of the subactions will be combined in the composite action

Combinator Example

allocate a cell **then** bind “M” to the given cell

- **then** is a combinator that passes the outgoing transients of the first sub-action to the incoming transients of the second sub-action

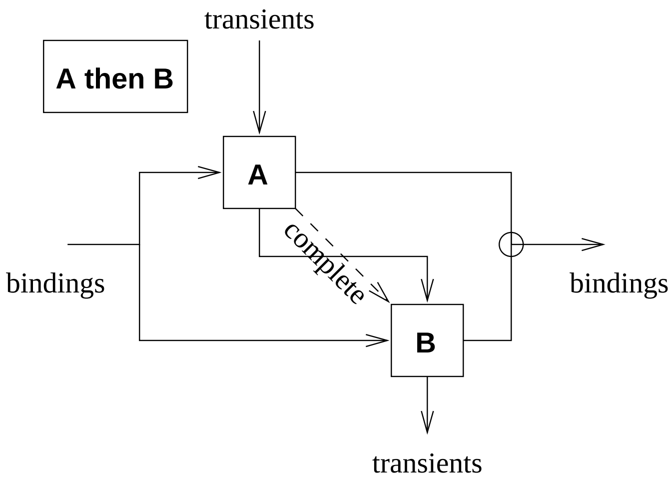
- There are many ways of propogating transients, bindings, and storage among the subactions of an action.

- If we restrict ourselves to deterministic actions, then storage is single threaded and can be treated as a global store

- To understand combinators we can look at how the transients and bindings are propogated

Combinator Diagrams

Devised by Ken Slonneger, they help us visualize how transients and bindings are propogated between the sub-actions of a combinator



Execution proceeds from the upper left to the lower right

Overview Notes

Slow Down.

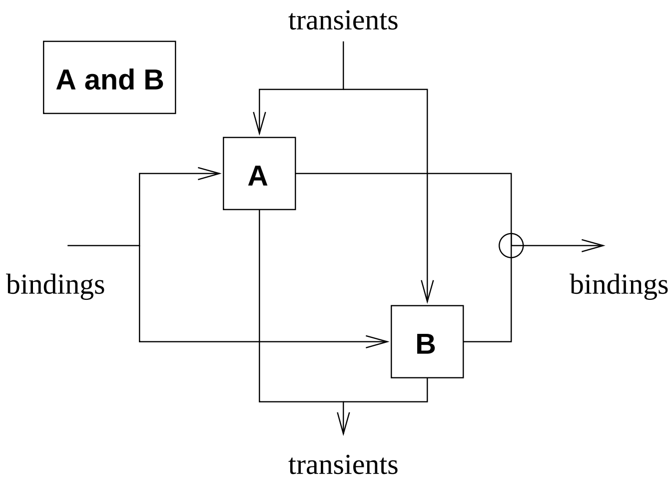
In this picture sub-action A receives the transients of the combined action. A's outgoing transients become B's incoming transients.

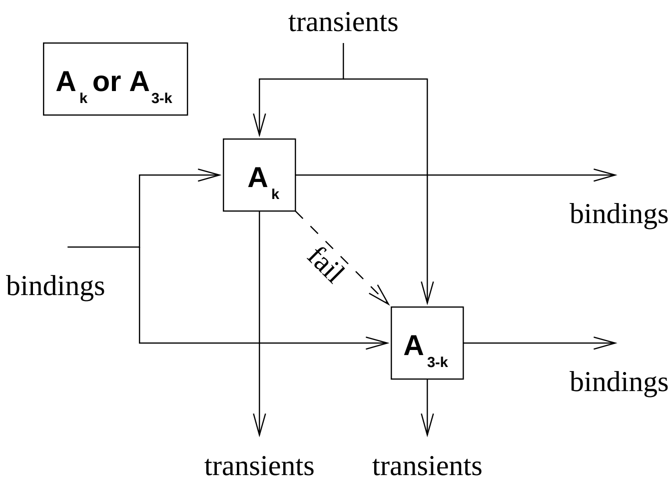
B's outgoing transients are the outgoing transients of the combined action

The bindings passed to the combined action are passed on to the subactions.

The bindings produced by A and B are merged to produce the bindings of the combined action.

Some Other Combinators





Action Semantic Definitions

- Like other semantic formalisms we wish to construct a mapping from the concrete syntax of a language to its semantics (actions).

- My implementation skips the abstract syntax representation

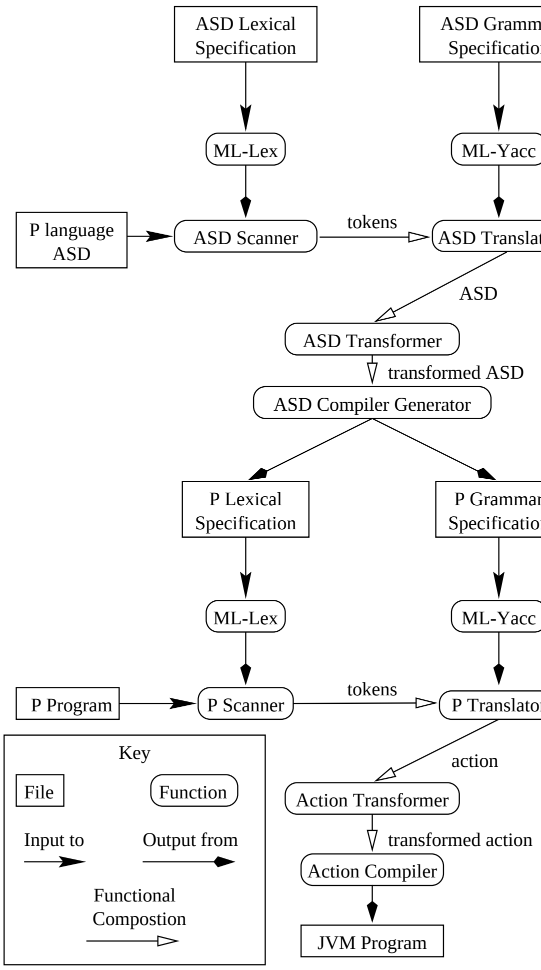
- Action Semantic Descriptions are modular.

- They are easier to extend than say a denotational description

Put up the calculator asd

Automated Compiler Generation

Using an Action Semantic Description (ASD) it's straightforward to write a parser generator that given an ASD and a source program will map the program to it's action



Calculator Program

- For example, consider the program 4+5=
- It can be mapped to its action using the ASD above.
- Its action is:

Put up the slide on the 4+5= program

Problem

- We want to generate efficient code from the program action

- Actions require bindings and transients to be passed from action to action

- Passing transients and bindings at run-time would be costly

- Traditional compilers for statically bound languages don't pass transients or bindings at run-time

- How should we eliminate transients and bindings from actions of statically bound languages?

Two Approaches

1. We could make the code generator very smart and have it work at eliminating the transients and bindings. This approach was used by Peter Orbaek in his compiler generator, Oasis.

2. Or we could take the approach put forward by Watt, Brown, and Moura. They perform sort inference on the program's action and use the information obtained to eliminate the transients and bindings.

Action Sort Inference

- Based on Record Type Inference described by David Schmidt and Susan Even.

- Deryck Brown extended it to work in his Actress Compiler Generator

- Records are maps from identifiers to sorts

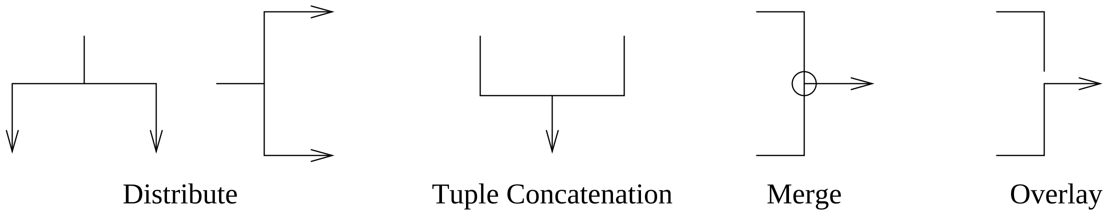
- Four records, one for each of the incoming and outgoing transients and bindings, describe the sort of an action.

- Storage is ignored since it's domain can't be known statically

- Brown defines five operators on Records that together implement sort inference over all action combinators

Record Sort Operations

- Distribute, Merge, Switch, Select, and Overlay are the operators Brown proposes. These operators correspond to the operations shown in the combinator diagrams above.



- The Actress Compiler Generator implements a dialect of Action Semantics that forces you to give transients in an odd way

- The Tuple Concatenation operator enables my compiler generator to represent Actions as described by Mosses in his text on the subject.

Sort Inference Example

- A complete example would be too tedious to cover here

- It should be sufficient to say this is one of those "left as an exercise for the reader" problems.

- After applying sort inference to our example above (i.e. 4+5=) we get the annotated action found here

Put up test0.sorts

Action Transformations

- By applying sort inference carefully it is possible to use the sort information to remove all transients and bindings from statically bound languages.

- In the example θ_0, θ_4, and θ_5 appear where ever the cell bound to "M", 4, or 5 appear respectively

- Moura describes these transformations in his thesis

Transient Elimination

- Transients can be eliminated by using sort information to rewrite yielders that give values based on the incoming transients.

- For instance, the action

  (give 4 and then give 5)

  then

  give sum(given integer#1,given integer#2)

- Can be rewritten as

  give sum(4,5)

Transient Elimination Algorithm

- Transient Elimination is achieved by replacing all give actions with the null action - complete

- Replace all given yielders with their actual values

  (complete and then complete)

  then

  given sum(4,5)

- Utilize algebraic identities of actions to simplify

  complete and then $a_2 = a_2$

  complete then $a_2 = a_2$

Binding Elimination

- Binding elimination is performed the same way transient elimination is performed

- Bind actions are replaced with complete

- Bound to yielders are replaced with their actual values

- Algebraic Identities involving complete are used to reduce the action

Conclusion

- Once Transient and Binding elimination have been performed, code generation is relatively easy.

- In fact, the actions that result from applying transient and binding elimination look a lot like assembly language code.

- Future work includes applying sort inference to Action Semantic Descriptions to validate and simplify *descriptions* of programming languages in addition to applying sort inference to each individual program's action