A Reference Interpreter for the Graph Programming Language

GP 2

Christopher Bak¹, Glyn Faulkner¹, Colin Runciman and Detlef Plump

Department of Computer Science, The University of York

Graphs as Models, April 2015

¹Supported by EPRSC Doctoral Training Grants.
Outline

- The GP 2 Language
  - The GP 2 Reference Interpreter
    - Motivation & Requirements
    - Implementation
    - Performance
  - Performance

- Conclusions and Future Work
Graph Programs

- Domain-specific language for graph-based structures
- User supplies the input graph and the graph transformation rules
- Small set of imperative control constructs to organise rule applications
- Non-deterministic execution
- Simple syntax and semantics to facilitate formal reasoning
Semantics

\[ \text{[Call}_1\text{]} \quad \frac{G \Rightarrow_R H}{\langle R, G \rangle \rightarrow H} \]
\[ \text{[Call}_2\text{]} \quad \frac{G \nRightarrow_R}{\langle R, G \rangle \rightarrow \text{fail}} \]
\[ \text{[Alap}_1\text{]} \quad \frac{\langle P, G \rangle \rightarrow^+ H}{\langle P!, G \rangle \rightarrow \langle P!, H \rangle} \]
\[ \text{[Alap}_2\text{]} \quad \frac{\langle P, G \rangle \rightarrow^+ \text{fail}}{\langle P!, G \rangle \rightarrow G} \]

Graphs G and H are states. \text{fail} is the failure state. R is a rule. P is a program.

\[1\text{ The Design of GP2, Detlef Plump, EPTCS 82 (2012)}\]
Transitive Closure

Main = link!

link(a,b,x,y,z: list)

where not edge(1, 3)

- Rule link applied as long as possible on the input graph.
- List labels used for generality.
Vertex Colouring

Main = init!; inc!
init(x: list)

Always outputs a valid colouring.
Minimal colouring not guaranteed because of non-determinism.
Vertex Colouring

Main = \textbf{init!}; \textbf{inc!}
init(x: list)

\[
\begin{align*}
\text{x} & \Rightarrow \text{x:1} \\
1 & \quad \Rightarrow \quad 2
\end{align*}
\]
Vertex Colouring

Main = \textbf{init!}; \textbf{inc!}

\textbf{init}(x: \text{list})

\[
\begin{align*}
x & \Rightarrow x:1 \\
1 & \quad 2
\end{align*}
\]
Vertex Colouring

Main = init!; inc!

inc(a,x,y:list; i:int)

```
x:i ⇒ x:i
```

```
y:i ⇒ y:i+1
```

![Graph diagram]
Vertex Colouring

Main = init!; inc!

inc(a,x,y:list; i:int)

\[
\begin{align*}
\text{Main} &= \text{init!; inc!} \\
\text{inc}(a,x,y:\text{list}; \ i:\text{int}) \\
\end{align*}
\]

\[
\begin{align*}
\text{inc}(1,x,2,3; \ i=1) &\Rightarrow \text{inc}(1,x,2,3; \ i=2) \\
\end{align*}
\]

\[
\begin{align*}
\text{Main} &= \text{init!; inc!} \\
\text{inc}(1,x,2,3; \ i=1) &\Rightarrow \text{inc}(1,x,2,3; \ i=2) \\
\end{align*}
\]
Vertex Colouring

Main = init!; inc!

inc(a,x,y:list; i:int)

1

\[
\begin{array}{ccc}
  x:i & \xrightarrow{a} & y:i \\
  1 & & 2 \\
  \Rightarrow & & \\
  x:i & \xrightarrow{a} & y:i+1 \\
  1 & & 2 \\
\end{array}
\]
Vertex Colouring

Main = init!; \textbf{inc}!

\texttt{inc}(a, x, y: list; i: int)

\begin{align*}
x: i & \xrightarrow{a} y: i \\
1 & \quad 2
\end{align*}

\Rightarrow

\begin{align*}
x: i & \xrightarrow{a} y: i + 1 \\
1 & \quad 2
\end{align*}
Vertex Colouring

Main = init!; \textbf{inc}!
\textbf{inc}(a,x,y:\text{list}; i:\text{int})

\begin{align*}
&\quad x:i \quad a \quad y:i \\
&\Rightarrow \quad x:i \quad a \quad y:i+1 \\
&\quad 1 \quad 2 \quad 1 \quad 2
\end{align*}
Motivation

- Test correctness of later compiled implementations
- Fully implement non-determinacy
- Familiarise language implementers with the semantics of GP 2
- Identify any gaps or ambiguities in the semantics
Motivation

- Test correctness of later compiled implementations
- Fully implement non-determinacy
- Familiarise language implementers with the semantics of GP 2
- Identify any gaps or ambiguities in the semantics

Simplicity is an over-riding aim

- Speed and memory use are secondary concerns
- Sophistication is to be actively avoided if it complicates the implementation!
Requirements

General requirements:

- Quick to develop
- Easy to maintain and reason about
- Must be fast enough to do “useful work”
Requirements

General requirements:

- Quick to develop
- Easy to maintain and reason about
- Must be fast enough to do “useful work”

For a given program/host-graph pair...

- Generate all possible output graphs
- Produce all distinct output graphs up to isomorphism
- Output a single result
Requirements

Also, stand-alone tools:

- Isomorphism checker
- Host-graph generator
- Graph viewer
GP 2

Implementation

- Based on the GP 2 semantics
- Written in Haskell\(^2\)

\(^2\)https://www.haskell.org/
Approx. 1000 SLOC

Exploits distinctive features of Haskell to achieve conciseness:
- list-comprehensions
- lazy evaluation
Performance

- Produce a fourth-generation Sierpinski triangle in 6.5 seconds
- A cyclic graph of 1000 nodes fails an acyclicity test in 1.8 seconds
- Transitive closure of a linear graph of 50 nodes takes 66 seconds
- Vertex colouring a 9x9 grid in one-result mode takes less than a second . . .
- . . . but in all-result mode exceeds 5 minutes with only a 3x3 grid

A more detailed discussion of performance can be found in the paper
Conclusions

- We have developed a useful reference tool for our ongoing research
- Also useful ancillary tools:
  - GraphViz-based graph visualiser
  - Stand-alone isomorphism checker
  - Host-graph generator, based on hypergraph grammars
- Gained a clear understanding of the GP 2 semantics
- Become aware of some ‘edge-cases’ that might trip us up in our compiler work
Further work

- Better error reporting
- A performant compiler
- GUI program editor
- Formal verification against GP 2 semantics.