50th Anniversary of the Birth of Prolog:

Some reflections on Prolog's Evolution, Status, and Future

Manuel Hermenegildo[1,2]

ICLP'22 (FLoC'22), August 4, 2022

[1]T. U. of Madrid (UPM)
[2]IMDEA Software Institute

*Part of the contents of this talk appear in the recent TPLP paper "***50 years of Prolog and Beyond***," by*

Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V. Hermenegildo, Jose F. Morales, Jan Wielemaker, Daniel Diaz, Salvador Abreu, and Giovanni Ciatto

written for Prolog's 50th anniversary and TPLP's 20th anniversary.

The Year of Prolog

- Summer of 1972:
  Alain Colmerauer and team in Marseille develop the first version of Prolog.
- This event + earlier and later collaborations w/Bob Kowalski and colleagues in Edinburgh, lay the foundations for the Prolog and LP of today.
- The "Year of Prolog" celebrates the 50th anniversary of these events.
  Organizers: Association for Logic Programming and Prolog Heritage Association.
- Objectives:
  - Highlight the continuing significance of Prolog and LP for both symbolic, explainable AI, and computing more generally.
  - Inspire a new generation of students, by drawing their attention to the logic-based approach to computing.

The Year of Prolog

- Summer of 1972:
  Alain Colmerauer and team in Marseille develop the first version of Prolog.
- This event + earlier and later collaborations w/Bob Kowalski and colleagues in Edinburgh, lay the foundations for the Prolog and LP of today.
- The "Year of Prolog" celebrates the 50th anniversary of these events.
  Organizers: Association for Logic Programming and Prolog Heritage Association.
- Objectives:
  - Highlight the continuing significance of Prolog and LP for both symbolic, explainable AI, and computing more generally.
  - Inspire a new generation of students, by drawing their attention to the logic-based approach to computing.

The Year of Prolog

- Summer of 1972:
  Alain Colmerauer and team in Marseille develop the first version of Prolog.
- This event + earlier and later collaborations w/Bob Kowalski and colleagues in Edinburgh, lay the foundations for the Prolog and LP of today.

- The "Year of Prolog" celebrates the 50th anniversary of these events.
  Organizers: Association for Logic Programming and Prolog Heritage Association.

- Objectives:
  - Highlight the continuing significance of Prolog and LP for both symbolic, explainable AI, and computing more generally.
  - Inspire a new generation of students, by drawing their attention to the logic-based approach to computing.

The Year of Prolog

- Summer of 1972:
  Alain Colmerauer and team in Marseille develop the first version of Prolog.
- This event + earlier and later collaborations w/Bob Kowalski and colleagues in Edinburgh, lay the foundations for the Prolog and LP of today.
- The "Year of Prolog" celebrates the 50th anniversary of these events.
  Organizers: Association for Logic Programming and Prolog Heritage Association.
- Objectives:
  - Highlight the continuing significance of Prolog and LP for both symbolic, explainable AI, and computing more generally.
  - Inspire a new generation of students, by drawing their attention to the logic-based approach to computing.

The Year of Prolog

- Initiatives:
  - **ALP Alain Colmerauer Prolog Heritage Prize**. *For recent practical accomplishments that highlight the benefits of Prolog-inspired computing for the future.* Deadline: September 2
  - **Prolog Day Symposium** (November 10, 2022) in which the Alain Colmerauer Prize will be awarded (subsequent editions at ICLP).
  - **Prolog Education initiative** (long-term initiative):
    - map and provide Prolog education resources for educators,
    - introduce schoolchildren/young adults to logic, programming, and AI w/Prolog.
  - **Survey paper on Fifty Years of Prolog and Beyond** published in the 20th anniversary special issue of TPLP.
  - **Special sessions and invited talks** at conferences (e.g., CILC, ICLP/FLoC!).

and others... do join in! prologyear.logicprogramming.org

Activities are overseen by a Scientific Committee, chaired by Bob Kowalski.

The Year of Prolog

- Initiatives:
  - **ALP Alain Colmerauer Prolog Heritage Prize**. *For recent practical accomplishments that highlight the benefits of Prolog-inspired computing for the future.* Deadline: September 2
  - **Prolog Day Symposium** (November 10, 2022) in which the Alain Colmerauer Prize will be awarded (subsequent editions at ICLP).
  - **Prolog Education initiative** (long-term initiative):
    - map and provide Prolog education resources for educators,
    - introduce schoolchildren/young adults to logic, programming, and AI w/Prolog.
  - **Survey paper on Fifty Years of Prolog and Beyond** published in the 20th anniversary special issue of TPLP.
  - **Special sessions and invited talks** at conferences (e.g., CILC, ICLP/FLoC!).

and others... do join in! prologyear.logicprogramming.org

Activities are overseen by a Scientific Committee, chaired by Bob Kowalski.

The Year of Prolog

- Initiatives:
  - **ALP Alain Colmerauer Prolog Heritage Prize**. *For recent practical accomplishments that highlight the benefits of Prolog-inspired computing for the future.* Deadline: September 2
  - **Prolog Day Symposium** (November 10, 2022) in which the Alain Colmerauer Prize will be awarded (subsequent editions at ICLP).
  - **Prolog Education initiative** (long-term initiative):
    - map and provide Prolog education resources for educators,
    - introduce schoolchildren/young adults to logic, programming, and AI w/Prolog.
  - **Survey paper on Fifty Years of Prolog and Beyond** published in the 20th anniversary special issue of TPLP.
  - **Special sessions and invited talks** at conferences (e.g., CILC, ICLP/FLoC!).

and others... do join in! prologyear.logicprogramming.org

Activities are overseen by a Scientific Committee, chaired by Bob Kowalski.

The Year of Prolog

- Initiatives:
  - **ALP Alain Colmerauer Prolog Heritage Prize**. *For recent practical accomplishments that highlight the benefits of Prolog-inspired computing for the future.* Deadline: September 2
  - **Prolog Day Symposium** (November 10, 2022) in which the Alain Colmerauer Prize will be awarded (subsequent editions at ICLP).
  - **Prolog Education initiative** (long-term initiative):
    - map and provide Prolog education resources for educators,
    - introduce schoolchildren/young adults to logic, programming, and AI w/Prolog.
  - **Survey paper on Fifty Years of Prolog and Beyond** published in the 20th anniversary special issue of TPLP.
  - **Special sessions and invited talks** at conferences (e.g., CILC, ICLP/FLoC!).

and others... do join in! prologyear.logicprogramming.org

Activities are overseen by a Scientific Committee, chaired by Bob Kowalski.

The Year of Prolog

- Initiatives:
  - **ALP Alain Colmerauer Prolog Heritage Prize**. *For recent practical accomplishments that highlight the benefits of Prolog-inspired computing for the future.* Deadline: September 2
  - **Prolog Day Symposium** (November 10, 2022) in which the Alain Colmerauer Prize will be awarded (subsequent editions at ICLP).
  - **Prolog Education initiative** (long-term initiative):
    - map and provide Prolog education resources for educators,
    - introduce schoolchildren/young adults to logic, programming, and AI w/Prolog.
  - **Survey paper on Fifty Years of Prolog and Beyond** published in the 20th anniversary special issue of TPLP.
  - **Special sessions and invited talks** at conferences (e.g., CILC, ICLP/FLoC!).

and others... do join in! prologyear.logicprogramming.org

Activities are overseen by a Scientific Committee, chaired by Bob Kowalski.

The Year of Prolog

- Initiatives:
  - **ALP Alain Colmerauer Prolog Heritage Prize**. *For recent practical accomplishments that highlight the benefits of Prolog-inspired computing for the future.* Deadline: September 2
  - **Prolog Day Symposium** (November 10, 2022) in which the Alain Colmerauer Prize will be awarded (subsequent editions at ICLP).
  - **Prolog Education initiative** (long-term initiative):
    - map and provide Prolog education resources for educators,
    - introduce schoolchildren/young adults to logic, programming, and AI w/Prolog.
  - **Survey paper on Fifty Years of Prolog and Beyond** published in the 20th anniversary special issue of TPLP.
  - **Special sessions and invited talks** at conferences (e.g., CILC, ICLP/FLoC!).

and others... do join in! prologyear.logicprogramming.org

Activities are overseen by a Scientific Committee, chaired by Bob Kowalski.

The Year of Prolog

- Initiatives:
  - **ALP Alain Colmerauer Prolog Heritage Prize**. *For recent practical accomplishments that highlight the benefits of Prolog-inspired computing for the future.* Deadline: September 2
  - **Prolog Day Symposium** (November 10, 2022) in which the Alain Colmerauer Prize will be awarded (subsequent editions at ICLP).
  - **Prolog Education initiative** (long-term initiative):
    - map and provide Prolog education resources for educators,
    - introduce schoolchildren/young adults to logic, programming, and AI w/Prolog.
  - **Survey paper on Fifty Years of Prolog and Beyond** published in the 20th anniversary special issue of TPLP.
  - **Special sessions and invited talks** at conferences (e.g., CILC, ICLP/FLoC!).

and others... do join in! prologyear.logicprogramming.org

Activities are overseen by a Scientific Committee, chaired by Bob Kowalski.

The Year of Prolog

- Initiatives:
  - **ALP Alain Colmerauer Prolog Heritage Prize**. *For recent practical accomplishments that highlight the benefits of Prolog-inspired computing for the future.* Deadline: September 2
  - **Prolog Day Symposium** (November 10, 2022) in which the Alain Colmerauer Prize will be awarded (subsequent editions at ICLP).
  - **Prolog Education initiative** (long-term initiative):
    - map and provide Prolog education resources for educators,
    - introduce schoolchildren/young adults to logic, programming, and AI w/Prolog.
  - **Survey paper on Fifty Years of Prolog and Beyond** published in the 20th anniversary special issue of TPLP.
  - **Special sessions and invited talks** at conferences (e.g., CILC, ICLP/FLoC!).

and others... do join in! prologyear.logicprogramming.org

Activities are overseen by a Scientific Committee, chaired by Bob Kowalski.

The Year of Prolog

- Initiatives:
  - **ALP Alain Colmerauer Prolog Heritage Prize**. *For recent practical accomplishments that highlight the benefits of Prolog-inspired computing for the future.* Deadline: September 2
  - **Prolog Day Symposium** (November 10, 2022) in which the Alain Colmerauer Prize will be awarded (subsequent editions at ICLP).
  - **Prolog Education initiative** (long-term initiative):
    - map and provide Prolog education resources for educators,
    - introduce schoolchildren/young adults to logic, programming, and AI w/Prolog.
  - **Survey paper on Fifty Years of Prolog and Beyond** published in the 20th anniversary special issue of TPLP.
  - **Special sessions and invited talks** at conferences (e.g., CILC, ICLP/FLoC!).

and others... do join in!

prologyear.logicprogramming.org

Activities are overseen by a Scientific Committee, chaired by Bob Kowalski.

- So, Prolog is 50!
  - What, 50 years?!? Half a century?!?!
  - Is Prolog therefore now 'old'? Is Prolog now irrelevant?

- Actually... continued interest:
  - Many *active implementations*, and *more appearing* continuously.
  - TIOBE index of programming languages shows Prolog:
    - In upper 10% of all languages tracked (270).
    - Stable, even somewhat upward trend since 2012.
    - One of only 13 languages that are tracked 'long term'.
  - A truly impressive body of research and scientific firsts.

- So, Prolog is 50!
  - What, 50 years?!? Half a century?!?!
  - Is Prolog therefore now 'old'? Is Prolog now irrelevant?

- Actually... continued interest:
  - Many *active implementations*, and *more appearing* continuously.
  - TIOBE index of programming languages shows Prolog:
    - In upper 10% of all languages tracked (270).
    - Stable, even somewhat upward trend since 2012.
    - One of only 13 languages that are tracked 'long term'.
  - A truly impressive body of research and scientific firsts.

- So, Prolog is 50!
  - What, 50 years?!? Half a century?!?!
  - Is Prolog therefore now 'old'? Is Prolog now irrelevant?
- Actually... continued interest:
  - Many *active implementations*, and *more appearing* continuously.
  - TIOBE index of programming languages shows Prolog:
    - In upper 10% of all languages tracked (270).
    - Stable, even somewhat upward trend since 2012.
    - One of only 13 languages that are tracked 'long term'.
  - A truly impressive body of research and scientific firsts.

- So, Prolog is 50!
  - What, 50 years?!? Half a century?!?!
  - Is Prolog therefore now 'old'? Is Prolog now irrelevant?
- Actually... continued interest:
  - Many *active implementations*, and *more appearing* continuously.
  - TIOBE index of programming languages shows Prolog:
    - In upper 10% of all languages tracked (270).
    - Stable, even somewhat upward trend since 2012.
    - One of only 13 languages that are tracked 'long term'.
  - A truly impressive body of research and scientific firsts.

Early steps, major milestones

Ancestors and birth

- Not possible to do full justice in this talk!
  See Kowalski (1988, 2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.
- Anyway, some highlights:
  - McCarthy (1962): the AI language LISP → "very high-level languages."
  - Robinson (1965): resolution inference rule.
  - Elcock (1967): Aberdeen System ("AbSys").
  - Green (1969): extend resolution to answer questions in FO-logic (QA3).
  - Colmerauer (1970): Q-systems.
  - Kowalski and Kuehner (1971): SL-resolution (focused search).
  - Boyer and Moore (1972): structure sharing.

  → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).

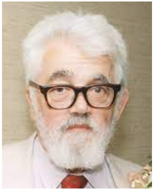
  → Prolog! (1972–1973)
  - The competing "procedural" view of AI (e.g., Hewitt).

  → Prompted Kowalski to marry the procedural and logical views.
  - Edinburgh: DHD Warren, +Pereira(s)/Bowen/Byrd; later Lisbon.

  → Dec-10 Prolog

...⟹

Ancestors and birth



- Not possible to do full justice in this talk!
  See Kowalski (1988, 2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.
- Anyway, some highlights:
  - McCarthy (1962): the AI language LISP → "very high-level languages."
  - Robinson (1965): resolution inference rule.
  - Elcock (1967): Aberdeen System ("AbSys").
  - Green (1969): extend resolution to answer questions in FO-logic (QA3).
  - Colmerauer (1970): Q-systems.
  - Kowalski and Kuehner (1971): SL-resolution (focused search).
  - Boyer and Moore (1972): structure sharing.
  - → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).
  - → Prolog! (1972–1973)
  - The competing "procedural" view of AI (e.g., Hewitt).
  - → Prompted Kowalski to marry the procedural and logical views.
  - Edinburgh: DHD Warren, +Pereira(s)/Bowen/Byrd; later Lisbon.
  - → Dec-10 Prolog

. . . ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  See Kowalski (1988, 2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.
- Anyway, some highlights:
  - McCarthy (1962): the AI language LISP → "very high-level languages."
  - Robinson (1965): resolution inference rule.
  - Elcock (1967): Aberdeen System ("AbSys").
  - Green (1969): extend resolution to answer questions in FO-logic (QA3).
  - Colmerauer (1970): Q-systems.
  - Kowalski and Kuehner (1971): SL-resolution (focused search).
  - Boyer and Moore (1972): structure sharing.
  - → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).
  - → Prolog! (1972–1973)
  - The competing "procedural" view of AI (e.g., Hewitt).
  - → Prompted Kowalski to marry the procedural and logical views.
  - Edinburgh: DHD Warren, +Pereira(s)/Bowen/Byrd; later Lisbon.
  - → Dec-10 Prolog

. . . ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  See Kowalski (1988, 2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.

- Anyway, some highlights:
  - McCarthy (1962): the AI language LISP → "very high-level languages."
  - Robinson (1965): resolution inference rule.
  - Elcock (1967): Aberdeen System ("AbSys").
  - Green (1969): extend resolution to answer questions in FO-logic (QA3).
  - Colmerauer (1970): Q-systems.
  - Kowalski and Kuehner (1971): SL-resolution (focused search).
  - Boyer and Moore (1972): structure sharing.
  - → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).
  - → Prolog! (1972–1973)
  - The competing "procedural" view of AI (e.g., Hewitt).
  - → Prompted Kowalski to marry the procedural and logical views.
  - Edinburgh: DHD Warren, +Pereira(s)/Bowen/Byrd; later Lisbon.
  - → Dec-10 Prolog

... ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  See Kowalski (1988, 2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.

- Anyway, some highlights:
  - McCarthy (1962): the AI language LISP → "very high-level languages."
  - Robinson (1965): resolution inference rule.
  - Elcock (1967): Aberdeen System ("AbSys").
  - Green (1969): extend resolution to answer questions in FO-logic (QA3).
  - Colmerauer (1970): Q-systems.
  - Kowalski and Kuehner (1971): SL-resolution (focused search).
  - Boyer and Moore (1972): structure sharing.

  → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).

  → Prolog! (1972–1973)
  - The competing "procedural" view of AI (e.g., Hewitt).

  → Prompted Kowalski to marry the procedural and logical views.
  - Edinburgh: DHD Warren, +Pereira(s)/Bowen/Byrd; later Lisbon.

  → Dec-10 Prolog

. . . ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  See Kowalski (1988, 2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.

- Anyway, some highlights:
  - ▶ McCarthy (1962): the AI language LISP → "very high-level languages."
  - ▶ Robinson (1965): resolution inference rule.
  - ▶ Elcock (1967): Aberdeen System ("AbSys").
  - ▶ Green (1969): extend resolution to answer questions in FO-logic (QA3).
  - ▶ Colmerauer (1970): Q-systems.
  - ▶ Kowalski and Kuehner (1971): SL-resolution (focused search).
  - ▶ Boyer and Moore (1972): structure sharing.
  - → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).
  - → Prolog! (1972–1973)
  - ▶ The competing "procedural" view of AI (e.g., Hewitt).
  - → Prompted Kowalski to marry the procedural and logical views.
  - ▶ Edinburgh: DHD Warren, +Pereira(s)/Bowen/Byrd; later Lisbon.
  - → Dec-10 Prolog

. . . ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  See Kowalski (1988, 2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.
- Anyway, some highlights:
  - McCarthy (1962): the AI language LISP → "very high-level languages."
  - Robinson (1965): resolution inference rule.
  - Elcock (1967): Aberdeen System ("AbSys").
  - Green (1969): extend resolution to answer questions in FO-logic (QA3).
  - Colmerauer (1970): Q-systems.
  - Kowalski and Kuehner (1971): SL-resolution (focused search).
  - Boyer and Moore (1972): structure sharing.
  - → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).
  - → Prolog! (1972–1973)
  - The competing "procedural" view of AI (e.g., Hewitt).
  - → Prompted Kowalski to marry the procedural and logical views.
  - Edinburgh: DHD Warren, +Pereira(s)/Bowen/Byrd; later Lisbon.
  - → Dec-10 Prolog

. . . ⇒

Ancestors and birth

- Not possible to do full justice in this talk!
  See Kowalski (1988, 2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.
- Anyway, some highlights:
  - ▶ McCarthy (1962): the AI language LISP → "very high-level languages."
  - ▶ Robinson (1965): resolution inference rule.
  - ▶ Elcock (1967): Aberdeen System ("AbSys").
  - ▶ Green (1969): extend resolution to answer questions in FO-logic (QA3).
  - ▶ Colmerauer (1970): Q-systems.
  - ▶ Kowalski and Kuehner (1971): SL-resolution (focused search).
  - ▶ Boyer and Moore (1972): structure sharing.
  - → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).
  - → Prolog! (1972–1973)
  - ▶ The competing "procedural" view of AI (e.g., Hewitt).
  - → Prompted Kowalski to marry the procedural and logical views.
  - ▶ Edinburgh: DHD Warren, +Pereira(s)/Bowen/Byrd; later Lisbon.
  - → Dec-10 Prolog

. . . ⟹

Ancestors and birth


- Not possible to do full justice in this talk!
  See Kowalski (1988, 2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.
- Anyway, some highlights:
  - McCarthy (1962): the AI language LISP → "very high-level languages."
  - Robinson (1965): resolution inference rule.
  - Elcock (1967): Aberdeen System ("AbSys").
  - Green (1969): extend resolution to answer questions in FO-logic (QA3).
  - Colmerauer (1970): Q-systems.
  - Kowalski and Kuehner (1971): SL-resolution (focused search).
  - Boyer and Moore (1972): structure sharing.

  → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).

  → Prolog! (1972–1973)
  - The competing "procedural" view of AI (e.g., Hewitt).

  → Prompted Kowalski to marry the procedural and logical views.
  - Edinburgh: DHD Warren, +Pereira(s)/Bowen/Byrd; later Lisbon.

  → Dec-10 Prolog

... ⟹

Ancestors and birth

- Not possible to do full justice in this talk! See Kowalski (1988, 2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.
- Anyway, some highlights:
  - ▶ McCarthy (1962): the AI language LISP → "very high-level languages."
  - ▶ Robinson (1965): resolution inference rule.
  - ▶ Elcock (1967): Aberdeen System ("AbSys").
  - ▶ Green (1969): extend resolution to answer questions in FO-logic (QA3).
  - ▶ Colmerauer (1970): Q-systems.
  - ▶ Kowalski and Kuehner (1971): SL-resolution (focused search).
  - ▶ Boyer and Moore (1972): structure sharing.
  - → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).
  - → Prolog! (1972–1973)
  - ▶ The competing "procedural" view of AI (e.g., Hewitt).
  - → Prompted Kowalski to marry the procedural and logical views.
  - ▶ Edinburgh: DHD Warren, +Pereira(s)/Bowen/Byrd; later Lisbon.
  - → Dec-10 Prolog

. . . ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  See Kowalski (1988, 2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.
- Anyway, some highlights:
  - McCarthy (1962): the AI language LISP → "very high-level languages."
  - Robinson (1965): resolution inference rule.
  - Elcock (1967): Aberdeen System ("AbSys").
  - Green (1969): extend resolution to answer questions in FO-logic (QA3).
  - Colmerauer (1970): Q-systems.
  - Kowalski and Kuehner (1971): SL-resolution (focused search).
  - Boyer and Moore (1972): structure sharing.
  - → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).
  - → Prolog! (1972–1973)
  - The competing "procedural" view of AI (e.g., Hewitt).
  - → Prompted Kowalski to marry the procedural and logical views.
  - Edinburgh: DHD Warren, +Pereira(s)/Bowen/Byrd; later Lisbon.
  - → Dec-10 Prolog

. . . ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  See Kowalski (1988, 2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.
- Anyway, some highlights:
  - ▶ McCarthy (1962): the AI language LISP → "very high-level languages."
  - ▶ Robinson (1965): resolution inference rule.
  - ▶ Elcock (1967): Aberdeen System ("AbSys").
  - ▶ Green (1969): extend resolution to answer questions in FO-logic (QA3).
  - ▶ Colmerauer (1970): Q-systems.
  - ▶ Kowalski and Kuehner (1971): SL-resolution (focused search).
  - ▶ Boyer and Moore (1972): structure sharing.
  - → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).
  - → Prolog! (1972–1973)
  - ▶ The competing "procedural" view of AI (e.g., Hewitt).
  - → Prompted Kowalski to marry the procedural and logical views.
  - ▶ Edinburgh: DHD Warren, +Pereira(s)/Bowen/Byrd; later Lisbon.
  - → Dec-10 Prolog

. . . ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  See Kowalski (1988, 2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.

- Anyway, some highlights:
  - McCarthy (1962): the AI language LISP → "very high-level languages."
  - Robinson (1965): resolution inference rule.
  - Elcock (1967): Aberdeen System ("AbSys").
  - Green (1969): extend resolution to answer questions in FO-logic (QA3).
  - Colmerauer (1970): Q-systems.
  - Kowalski and Kuehner (1971): SL-resolution (focused search).
  - Boyer and Moore (1972): structure sharing.
  - → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).
  - → Prolog! (1972–1973)
  - The competing "procedural" view of AI (e.g., Hewitt).
  - → Prompted Kowalski to marry the procedural and logical views.
  - Edinburgh: DHD Warren, +Pereira(s)/Bowen/Byrd; later Lisbon.
  - → Dec-10 Prolog

. . . ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  See Kowalski (1988, 2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.
- Anyway, some highlights:
  - McCarthy (1962): the AI language LISP → "very high-level languages."
  - Robinson (1965): resolution inference rule.
  - Elcock (1967): Aberdeen System ("AbSys").
  - Green (1969): extend resolution to answer questions in FO-logic (QA3).
  - Colmerauer (1970): Q-systems.
  - Kowalski and Kuehner (1971): SL-resolution (focused search).
  - Boyer and Moore (1972): structure sharing.
  - → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).
  - → Prolog! (1972–1973)
  - The competing "procedural" view of AI (e.g., Hewitt).
  - → Prompted Kowalski to marry the procedural and logical views.
  - Edinburgh: DHD Warren, +Pereira(s)/Bowen/Byrd; later Lisbon.
  - → Dec-10 Prolog

. . . ⇒

Ancestors and birth

- Not possible to do full justice in this talk!
  See Kowalski (1988, 2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.

- Anyway, some highlights:
  - ▶ McCarthy (1962): the AI language LISP → "very high-level languages."
  - ▶ Robinson (1965): resolution inference rule.
  - ▶ Elcock (1967): Aberdeen System ("AbSys").
  - ▶ Green (1969): extend resolution to answer questions in FO-logic (QA3).
  - ▶ Colmerauer (1970): Q-systems.
  - ▶ Kowalski and Kuehner (1971): SL-resolution (focused search).
  - ▶ Boyer and Moore (1972): structure sharing.
  - → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).
  - → Prolog! (1972–1973)
  - ▶ The competing "procedural" view of AI (e.g., Hewitt).
  - → Prompted Kowalski to marry the procedural and logical views.
  - ▶ Edinburgh: DHD Warren, +Pereira(s)/Bowen/Byrd; later Lisbon.
  - → Dec-10 Prolog

... ⇒

Early Prologs and main milestones (≈ up to ISO)

- First *Prolog*(s): fundamental characteristics already there!

Early Prologs and main milestones (≈ up to ISO)

- First *Prolog*(s): fundamental characteristics already there!

- Dec-10 Prolog: *Compilation* (+ improved syntax, etc.)

→ performance (≈ lisp),
→ much more widespread use –but portability.

Early Prologs and main milestones (≈ up to ISO)

- First *Prolog*(s): fundamental characteristics already there!
- Dec-10 Prolog: *Compilation* (+ improved syntax, etc.)

→ performance (≈ lisp),
→ much more widespread use –but portability.

- CDL-Prolog, MU-Prolog, ..., C-Prolog: portability (but interpreter).

Early Prologs and main milestones (≈ up to ISO)

- First *Prolog*(s): fundamental characteristics already there!
- Dec-10 Prolog: *Compilation* (+ improved syntax, etc.)

→ performance (≈ lisp),

→ much more widespread use –but portability.

- CDL-Prolog, MU-Prolog, ..., C-Prolog: portability (but interpreter).
- The *WAM:* portability + speed... and implementation beauty.

Early Prologs and main milestones (≈ up to ISO)

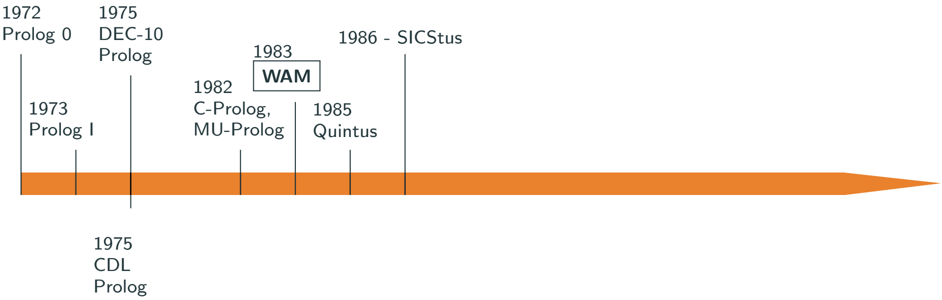
- First *Prolog*(s): fundamental characteristics already there!
- Dec-10 Prolog: *Compilation* (+ improved syntax, etc.)

→ performance (≈ lisp),
→ much more widespread use –but portability.

- CDL-Prolog, MU-Prolog, ..., C-Prolog: portability (but interpreter).
- The *WAM:* portability + speed... and implementation beauty.

(FGCS → MCC → ECRC → ESPRIT → EU research programs, and others.)

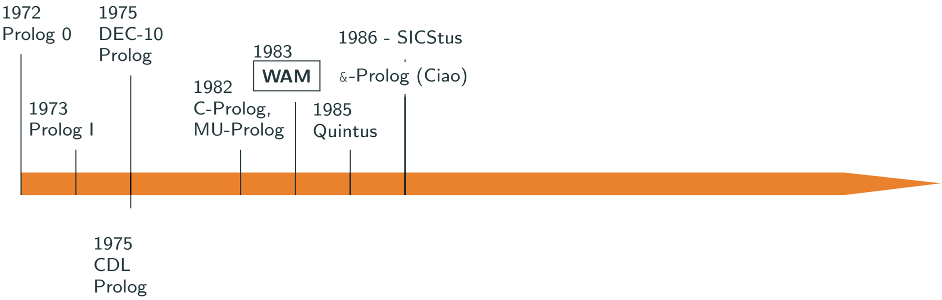
Early Prologs and main milestones (≈ up to ISO)



- WAM optimizations (Quintus, SICStus, BIM, YAP, ...), GC, ...

→ commercial/PD, dissemination, more performance.

Early Prologs and main milestones (≈ up to ISO)



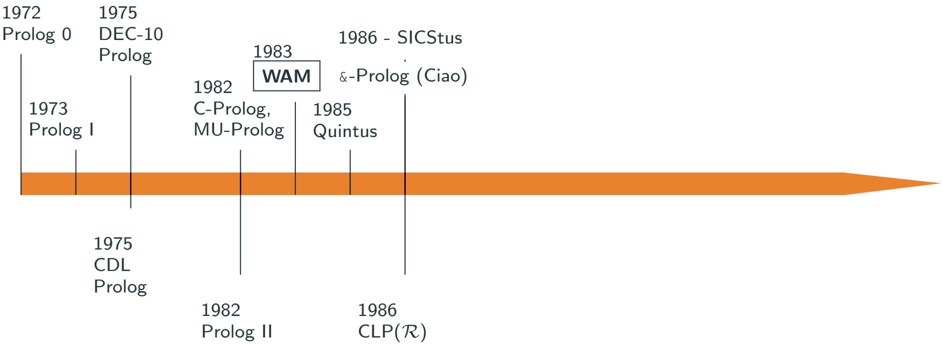
- WAM optimizations (Quintus, SICStus, BIM, YAP, ...), GC, ...

→ commercial/PD, dissemination, more performance.

- Or- and and-parallelism.
- *Global analysis* (abstract interpretation), P.Eval.; Aquarius, &-Prolog/Ciao.
  (Independence/aliasing, modes, types, determinacy, sharing, non-failure, cost, ...)
  First practical compiler(s) using abstract interpretation?

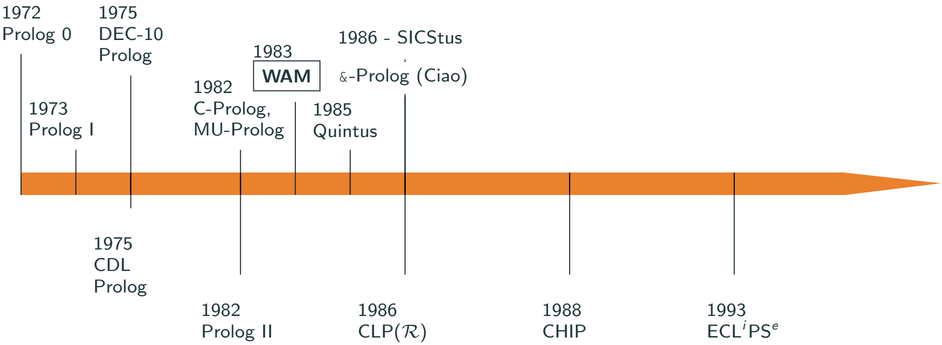
→ Performance (≈ imperative), auto-parallelization, real parallel speedups.

Early Prologs and main milestones (≈ up to ISO)



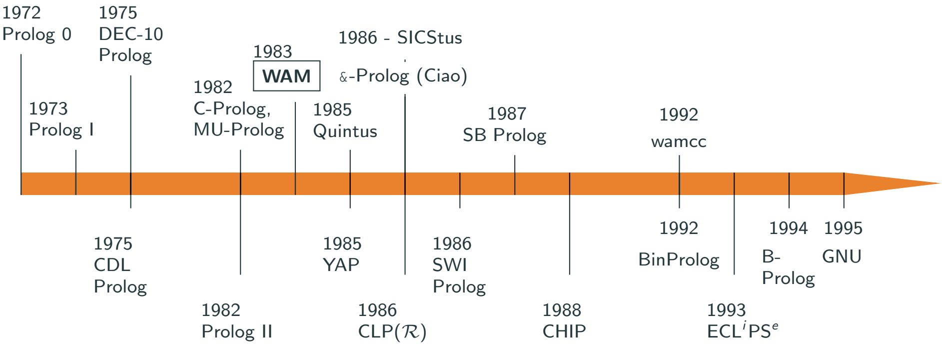
- *Constraints* (Prolog II, CLP scheme/CLP($\mathcal{R}$))

Early Prologs and main milestones (≈ up to ISO)



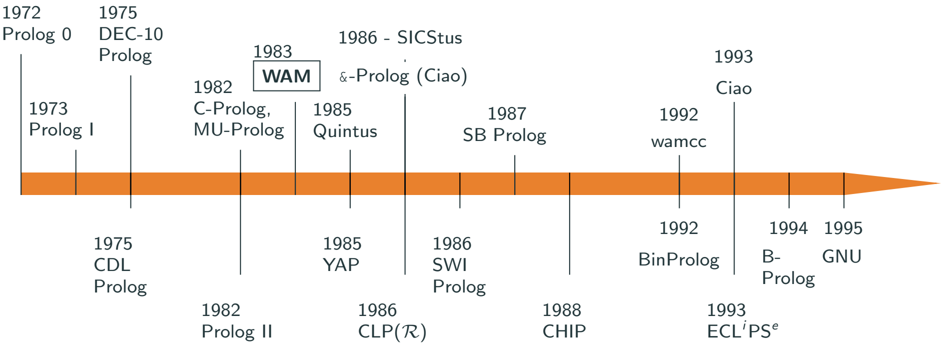
- *Constraints* (Prolog II, CLP scheme/CLP($\mathcal{R}$))
  - Finite domains.

Early Prologs and main milestones (≈ up to ISO)



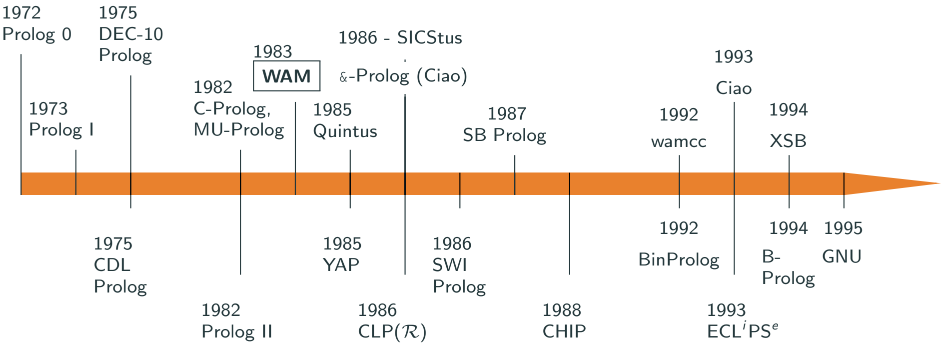
- *Constraints* (Prolog II, CLP scheme/CLP($\mathcal{R}$))
  - Finite domains.
- A good number of other WAM-(and non-WAM)-based Prologs (see later).

Early Prologs and main milestones (≈ up to ISO)



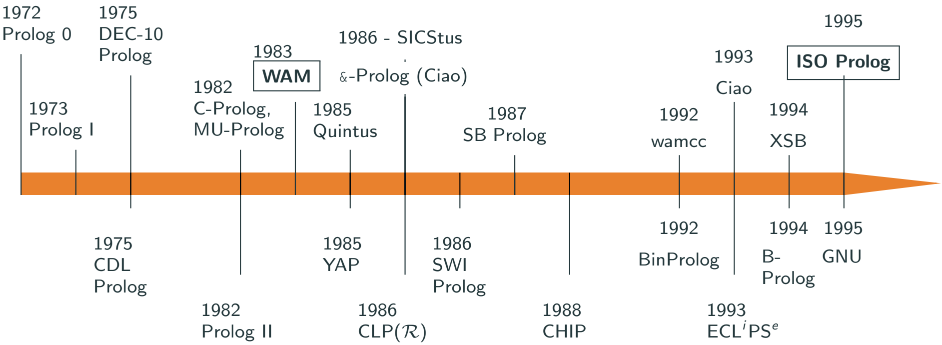
- *Constraints* (Prolog II, CLP scheme/CLP($\mathcal{R}$))
  - Finite domains.
- A good number of other WAM-(and non-WAM)-based Prologs (see later).
- Higher-order / functional syntax support (λ-Prolog, HiLog, Hiord, ...).
- *Types/modes*, verification, testing, assertions.

Early Prologs and main milestones (≈ up to ISO)



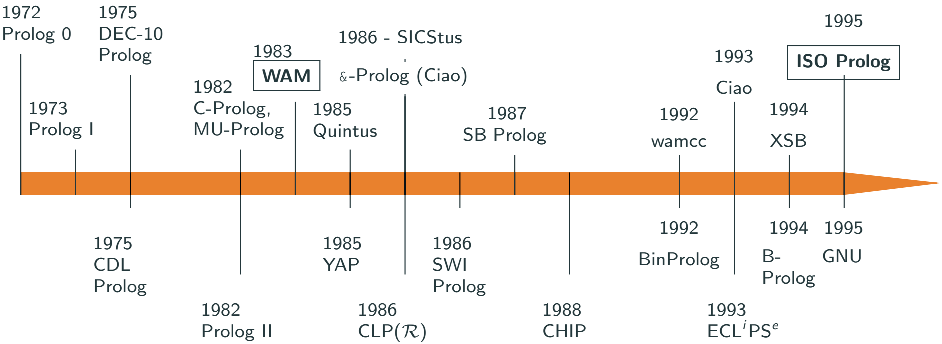
- *Constraints* (Prolog II, CLP scheme/CLP($\mathcal{R}$))
  - Finite domains.
- A good number of other WAM-(and non-WAM)-based Prologs (see later).
- Higher-order / functional syntax support (λ-Prolog, HiLog, Hiord, ...).
- *Types/modes*, verification, testing, assertions.
- Early ded., *Tabling*, SLG-resolution, minimal-model / well-founded semantics.

Early Prologs and main milestones ($\approx$ up to ISO)



- *Constraints* (Prolog II, CLP scheme/CLP($\mathcal{R}$))
  - Finite domains.
- A good number of other WAM-(and non-WAM)-based Prologs (see later).
- Higher-order / functional syntax support (λ-Prolog, HiLog, Hiord, ...).
- *Types/modes*, verification, testing, assertions.
- Early ded., *Tabling*, SLG-resolution, minimal-model / well-founded semantics.

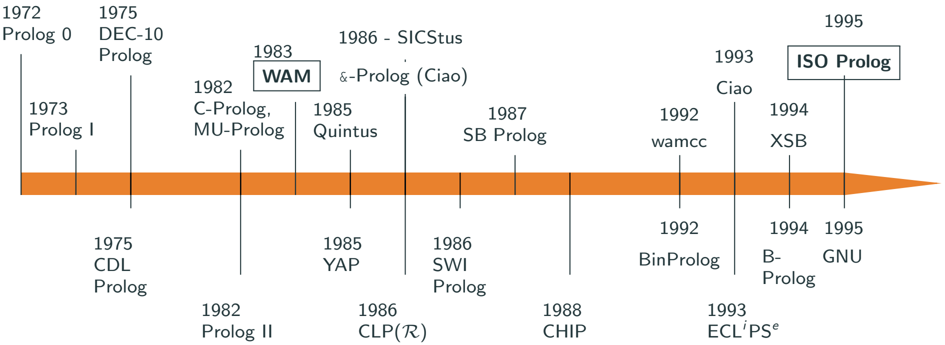
Early Prologs and main milestones (≈ up to ISO)



All this progressed in parallel with further advances in the theoretical underpinnings:

- Kowalski/van Emden (1976): linear res. for Horn clauses, no factoring rule, ...
- Clark (1978): correctness of NaF w.r.t. program completion.
- Reiter (1978): formalization of "Closed world assumption."
- Minker, Gallaire, Cohen, Lassez/Jaffar/Maher, DHD Warren, Tamaki/Sato, DS Warren, ...

Early Prologs and main milestones (≈ up to ISO)

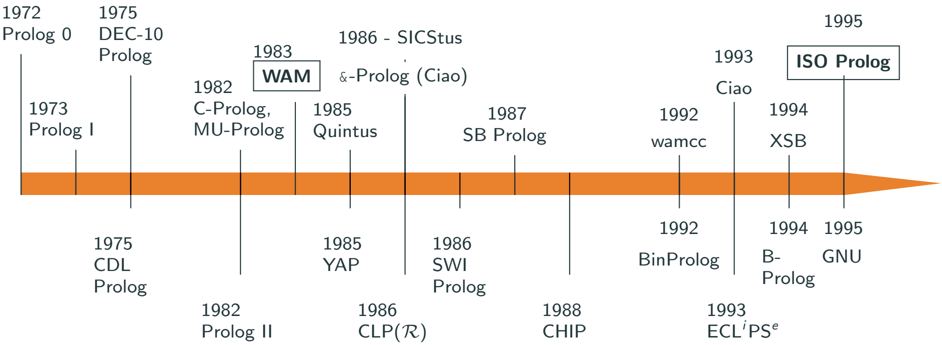


After ISO – much additional evolution:

- Constraints in standard Prologs: "Opening the box" (attvars/CHR).
- Learning (ILP), probabilistic.
- *ASP* ⇝ Prolog-ASP combinations ⇝ *s(CASP)*.
- Web embedding, playgrounds, notebooks.

\+ applications of techniques to other languages,
combination with deep learning / explainable AI, ...

Early Prologs and main milestones (≈ up to ISO)



After ISO – much additional evolution:

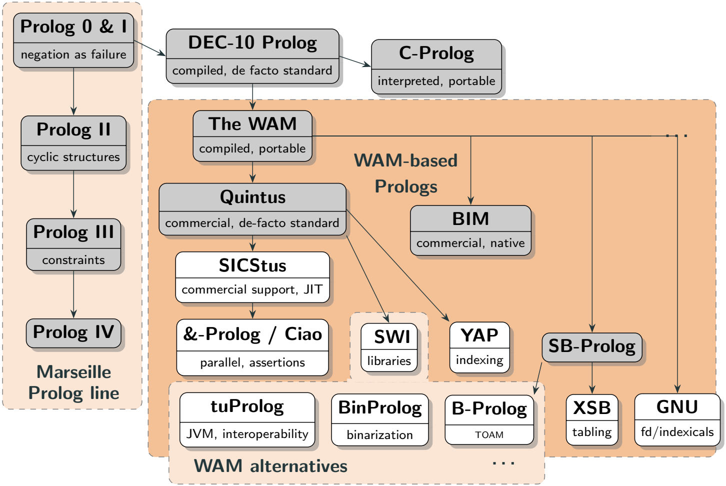
- Constraints in standard Prologs: "Opening the box" (attvars/CHR).
- Learning (ILP), probabilistic.
- *ASP* ⇝ Prolog-ASP combinations ⇝ *s(CASP)*.
- Web embedding, playgrounds, notebooks.

\+ applications of techniques to other languages,
combination with deep learning / explainable AI, ...

Let's jump forward and take a look at the current state of things!

An overview of current systems

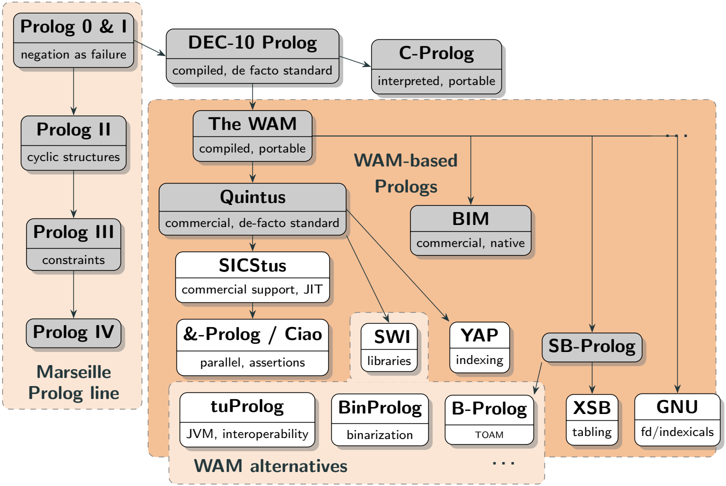
Prolog system heritage



White background: currently active/supported systems.
Lower legends: just some highlight(s) (see later).
Arrows: influences and inspiration.

Again, more missing!: ECL^{i}PS^e, IBM, LIFE, Andorra-I, Scryer, Tau, ...

Prolog system heritage



White background: currently active/supported systems.
Lower legends: just some highlight(s) (see later).
Arrows: influences and inspiration.

Again, more missing!: ECL^{i}PS^e, IBM, LIFE, Andorra-I, Scryer, Tau, ...

Support status for selected features - I

System	Open Src.	Modules	Non-Std. Data Types	Foreign Language Interfaces
B-Prolog			arrays, sets, hashtables	C, Java
Ciao	✓	✓		C, Java, Python, JScrpt
ECLiPSe	✓	✓	arrays, strings	C, Java, Python, PHP
GNU Prolog	✓		arrays	C, Java, PHP
JIProlog	✓	✓		Java
SICStus		✓		C, Java, .NET, Tcl/Tk
SWI	✓	✓	dicts, strings	C, C++, Java
τProlog	✓	✓		JavaScript
tuProlog	✓		arrays	Java, .NET, Android, iOS
XSB	✓	✓		C, Java, PERL, Python
YAP	✓	✓		C, Python, R

Support status for selected features - I

System	Open Src.	Modules	Non-Std. Data Types	Foreign Language Interfaces
B-Prolog			arrays, sets, hashtables	C, Java
Ciao	✓	✓		C, Java, Python, JScrpt
ECLiPSe	✓	✓	arrays, strings	C, Java, Python, PHP
GNU Prolog	✓		arrays	C, Java, PHP
JIProlog	✓	✓		Java
SICStus		✓		C, Java, .NET, Tcl/Tk
SWI	✓	✓	dicts, strings	C, C++, Java
τProlog	✓	✓		JavaScript
tuProlog	✓		arrays	Java, .NET, Android, iOS
XSB	✓	✓		C, Java, PERL, Python
YAP	✓	✓		C, Python, R

Support status for selected features - I

System	Open Src.	Modules	Non-Std. Data Types	Foreign Language Interfaces
B-Prolog			arrays, sets, hashtables	C, Java
Ciao	✓	✓		C, Java, Python, JScrpt
ECLiPSe	✓	✓	arrays, strings	C, Java, Python, PHP
GNU Prolog	✓		arrays	C, Java, PHP
JIProlog	✓	✓		Java
SICStus		✓		C, Java, .NET, Tcl/Tk
SWI	✓	✓	dicts, strings	C, C++, Java
τProlog	✓	✓		JavaScript
tuProlog	✓		arrays	Java, .NET, Android, iOS
XSB	✓	✓		C, Java, PERL, Python
YAP	✓	✓		C, Python, R

Support status for selected features - I

System	Open Src.	Modules	Non-Std. Data Types	Foreign Language Interfaces
B-Prolog			arrays, sets, hashtables	C, Java
Ciao	✓	✓		C, Java, Python, JScrpt
ECLiPSe	✓	✓	arrays, strings	C, Java, Python, PHP
GNU Prolog	✓		arrays	C, Java, PHP
JIProlog	✓	✓		Java
SICStus		✓		C, Java, .NET, Tcl/Tk
SWI	✓	✓	dicts, strings	C, C++, Java
τProlog	✓	✓		JavaScript
tuProlog	✓		arrays	Java, .NET, Android, iOS
XSB	✓	✓		C, Java, PERL, Python
YAP	✓	✓		C, Python, R

Support status for selected features - I

System	Open Src.	Modules	Non-Std. Data Types	Foreign Language Interfaces
B-Prolog			arrays, sets, hashtables	C, Java
Ciao	✓	✓		C, Java, Python, JScrpt
ECLiPSe	✓	✓	arrays, strings	C, Java, Python, PHP
GNU Prolog	✓		arrays	C, Java, PHP
JIProlog	✓	✓		Java
SICStus		✓		C, Java, .NET, Tcl/Tk
SWI	✓	✓	dicts, strings	C, C++, Java
τProlog	✓	✓		JavaScript
tuProlog	✓		arrays	Java, .NET, Android, iOS
XSB	✓	✓		C, Java, PERL, Python
YAP	✓	✓		C, Python, R

Support status for selected features - II

System	CLP	CHR	Tabling	Parallelism	Indexing	Coroutines
B-Prolog	$\mathcal{FD}$, $\mathcal{B}$, *Set*	✓	✓		N-FA	✓
Ciao	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	FA, MA	✓
ECLiPSe	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$, *Set*	✓		✓	most suitable	✓
GNU Prolog	$\mathcal{FD}$, $\mathcal{B}$				FA	
JIProlog					undocumented	
SICStus	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓			FA	✓
SWI	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	MA, deep, JIT	✓
τProlog					undocumented	
tuProlog				✓	FA	
XSB	$\mathcal{R}$	✓	✓	✓	all, trie	✓
YAP	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓		FA, MA, JIT	

Support status for selected features - II

System	CLP	CHR	Tabling	Parallelism	Indexing	Coroutines
B-Prolog	$\mathcal{FD}$, $\mathcal{B}$, *Set*	✓	✓		N-FA	✓
Ciao	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	FA, MA	✓
ECLiPSe	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$, *Set*	✓		✓	most suitable	✓
GNU Prolog	$\mathcal{FD}$, $\mathcal{B}$				FA	
JIProlog					undocumented	
SICStus	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓			FA	✓
SWI	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	MA, deep, JIT	✓
τProlog					undocumented	
tuProlog				✓	FA	
XSB	$\mathcal{R}$	✓	✓	✓	all, trie	✓
YAP	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓		FA, MA, JIT	

Support status for selected features - II

System	CLP	CHR	Tabling	Parallelism	Indexing	Coroutines
B-Prolog	$\mathcal{FD}$, $\mathcal{B}$, *Set*	✓	✓		N-FA	✓
Ciao	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	FA, MA	✓
ECLiPSe	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$, *Set*	✓		✓	most suitable	✓
GNU Prolog	$\mathcal{FD}$, $\mathcal{B}$				FA	
JIProlog					undocumented	
SICStus	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓			FA	✓
SWI	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	MA, deep, JIT	✓
τProlog					undocumented	
tuProlog				✓	FA	
XSB	$\mathcal{R}$	✓	✓	✓	all, trie	✓
YAP	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓		FA, MA, JIT	

Support status for selected features - II

System	CLP	CHR	Tabling	Parallelism	Indexing	Coroutines
B-Prolog	$\mathcal{FD}$, $\mathcal{B}$, *Set*	✓	✓		N-FA	✓
Ciao	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	FA, MA	✓
ECLiPSe	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$, *Set*	✓		✓	most suitable	✓
GNU Prolog	$\mathcal{FD}$, $\mathcal{B}$				FA	
JIProlog					undocumented	
SICStus	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓			FA	✓
SWI	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	MA, deep, JIT	✓
τProlog					undocumented	
tuProlog				✓	FA	
XSB	$\mathcal{R}$	✓	✓	✓	all, trie	✓
YAP	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓		FA, MA, JIT	

Support status for selected features - II

System	CLP	CHR	Tabling	Parallelism	Indexing	Coroutines
B-Prolog	$\mathcal{FD}$, $\mathcal{B}$, *Set*	✓	✓		N-FA	✓
Ciao	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	FA, MA	✓
ECLiPSe	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$, *Set*	✓		✓	most suitable	✓
GNU Prolog	$\mathcal{FD}$, $\mathcal{B}$				FA	
JIProlog					undocumented	
SICStus	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓			FA	✓
SWI	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	MA, deep, JIT	✓
τProlog					undocumented	
tuProlog				✓	FA	
XSB	$\mathcal{R}$	✓	✓	✓	all, trie	✓
YAP	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓		FA, MA, JIT	

Support status for selected features - II

System	CLP	CHR	Tabling	Parallelism	Indexing	Coroutines
B-Prolog	$\mathcal{FD}$, $\mathcal{B}$, *Set*	✓	✓		N-FA	✓
Ciao	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	FA, MA	✓
ECLiPSe	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$, *Set*	✓		✓	most suitable	✓
GNU Prolog	$\mathcal{FD}$, $\mathcal{B}$				FA	
JIProlog					undocumented	
SICStus	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓			FA	✓
SWI	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	MA, deep, JIT	✓
τProlog					undocumented	
tuProlog				✓	FA	
XSB	$\mathcal{R}$	✓	✓	✓	all, trie	✓
YAP	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓		FA, MA, JIT	

Support status for selected features - III

System	Debugger	Global Vars.	Mutables	Testing	Types/Modes	s(CASP)
B-Prolog	trace	✓				
Ciao	trace / source	✓	✓	✓	✓	✓
ECLiPSe	trace	✓		✓		
GNU Prolog	trace	✓	✓			
JIProlog	trace					
SICStus	trace / source		✓	✓		
SWI	trace / graphical	✓	✓	✓		✓
τProlog						
tuProlog	spy					
XSB	trace					
YAP	trace	✓				

Support status for selected features - III

System	Debugger	Global Vars.	Mutables	Testing	Types/Modes	s(CASP)
B-Prolog	trace	✓				
Ciao	trace / source	✓	✓	✓	✓	✓
ECLiPSe	trace	✓		✓		
GNU Prolog	trace	✓	✓			
JIProlog	trace					
SICStus	trace / source		✓	✓		
SWI	trace / graphical	✓	✓	✓		✓
τProlog						
tuProlog	spy					
XSB	trace					
YAP	trace	✓				

Support status for selected features - III

System	Debugger	Global Vars.	Mutables	Testing	Types/Modes	s(CASP)
B-Prolog	trace	✓				
Ciao	trace / source	✓	✓	✓	✓	✓
ECLiPSe	trace	✓		✓		
GNU Prolog	trace	✓	✓			
JIProlog	trace					
SICStus	trace / source		✓	✓		
SWI	trace / graphical	✓	✓	✓		✓
τProlog						
tuProlog	spy					
XSB	trace					
YAP	trace	✓				

Support status for selected features - III

System	Debugger	Global Vars.	Mutables	Testing	Types/Modes	s(CASP)
B-Prolog	trace	✓				
Ciao	trace / source	✓	✓	✓	✓	✓
ECLiPSe	trace	✓		✓		
GNU Prolog	trace	✓	✓			
JIProlog	trace					
SICStus	trace / source		✓	✓		
SWI	trace / graphical	✓	✓	✓		✓
τProlog						
tuProlog	spy					
XSB	trace					
YAP	trace	✓				

Support status for selected features - III

System	Debugger	Global Vars.	Mutables	Testing	Types/Modes	s(CASP)
B-Prolog	trace	✓				
Ciao	trace / source	✓	✓	✓	✓	✓
ECLiPSe	trace	✓		✓		
GNU Prolog	trace	✓	✓			
JIProlog	trace					
SICStus	trace / source		✓	✓		
SWI	trace / graphical	✓	✓	✓		✓
τProlog						
tuProlog	spy					
XSB	trace					
YAP	trace	✓				

Support status for selected features - III

System	Debugger	Global Vars.	Mutables	Testing	Types/Modes	s(CASP)
B-Prolog	trace	✓				
Ciao	trace / source	✓	✓	✓	✓	✓
ECLiPSe	trace	✓		✓		
GNU Prolog	trace	✓	✓			
JIProlog	trace					
SICStus	trace / source		✓	✓		
SWI	trace / graphical	✓	✓	✓		✓
τProlog						
tuProlog	spy					
XSB	trace					
YAP	trace	✓				

Support status for selected features - III

System	Debugger	Global Vars.	Mutables	Testing	Types/Modes	s(CASP)
B-Prolog	trace	✓				
Ciao	trace / source	✓	✓	✓	✓	✓
ECLiPSe	trace	✓		✓		
GNU Prolog	trace	✓	✓			
JIProlog	trace					
SICStus	trace / source		✓	✓		
SWI	trace / graphical	✓	✓	✓		✓
τProlog						
tuProlog	spy					
XSB	trace					
YAP	trace	✓				

Many other features!

- Auto-documentation, attributed variables, objects, enhanced expansions, playgrounds, ...

Summary (so far)

- Prolog systems have come a long way!
- ISO standard generally supported (with minor differences).
- *Basic* module system pretty compatible.
- A good number of commonly available features:
  - Constraints.
  - Multi-threading.
  - Tabling.
  - Coroutining.
  - ...

  However,

  - Interfaces and details often differ.
    Can mostly be bridged (c.f., Paolo Moura's work), but a real nuisance.
  - Some features (e.g., Types/modes/verification, s(CASP), ...)
    still in few systems.

Summary (so far)

- Prolog systems have come a long way!
- ISO standard generally supported (with minor differences).
- *Basic* module system pretty compatible.
- A good number of commonly available features:
  - Constraints.
  - Multi-threading.
  - Tabling.
  - Coroutining.
  - ...

  However,
  - Interfaces and details often differ.
    Can mostly be bridged (c.f., Paolo Moura's work), but a real nuisance.
  - Some features (e.g., Types/modes/verification, s(CASP), ...)
    still in few systems.

Summary (so far)

- Prolog systems have come a long way!
- ISO standard generally supported (with minor differences).
- *Basic* module system pretty compatible.
- A good number of commonly available features:
  - Constraints.
  - Multi-threading.
  - Tabling.
  - Coroutining.
  - ...

  However,
  - Interfaces and details often differ.
    Can mostly be bridged (c.f., Paolo Moura's work), but a real nuisance.
  - Some features (e.g., Types/modes/verification, s(CASP), ...)
    still in few systems.

Summary (so far)

- Prolog systems have come a long way!
- ISO standard generally supported (with minor differences).
- *Basic* module system pretty compatible.
- A good number of commonly available features:
  - Constraints.
  - Multi-threading.
  - Tabling.
  - Coroutining.
  - ...

  However,
  - Interfaces and details often differ.
    Can mostly be bridged (c.f., Paolo Moura's work), but a real nuisance.
  - Some features (e.g., Types/modes/verification, s(CASP), ...)
    still in few systems.

Summary (so far)

- Prolog systems have come a long way!
- ISO standard generally supported (with minor differences).
- *Basic* module system pretty compatible.
- A good number of commonly available features:
  - Constraints.
  - Multi-threading.
  - Tabling.
  - Coroutining.
  - ...

However,

- Interfaces and details often differ.
  Can mostly be bridged (c.f., Paolo Moura's work), but a real nuisance.
- Some features (e.g., Types/modes/verification, s(CASP), ...)
  still in few systems.

Summary (so far)

- Prolog systems have come a long way!
- ISO standard generally supported (with minor differences).
- *Basic* module system pretty compatible.
- A good number of commonly available features:
  - Constraints.
  - Multi-threading.
  - Tabling.
  - Coroutining.
  - ...

  However,
  - Interfaces and details often differ.
    Can mostly be bridged (c.f., Paolo Moura's work), but a real nuisance.
  - Some features (e.g., Types/modes/verification, s(CASP), ...)
    still in few systems.

Summary (so far)

- Prolog systems have come a long way!
- ISO standard generally supported (with minor differences).
- *Basic* module system pretty compatible.
- A good number of commonly available features:
  - Constraints.
  - Multi-threading.
  - Tabling.
  - Coroutining.
  - ...

However,

- Interfaces and details often differ.
  Can mostly be bridged (c.f., Paolo Moura's work), but a real nuisance.
- Some features (e.g., Types/modes/verification, s(CASP), ...)
  still in few systems.

Summary (so far)

- Prolog systems have come a long way!
- ISO standard generally supported (with minor differences).
- *Basic* module system pretty compatible.
- A good number of commonly available features:
  - Constraints.
  - Multi-threading.
  - Tabling.
  - Coroutining.
  - ...

However,

- Interfaces and details often differ.
  Can mostly be bridged (c.f., Paolo Moura's work), but a real nuisance.
- Some features (e.g., Types/modes/verification, s(CASP), ...)
  still in few systems.

Summary (so far)

- Prolog systems have come a long way!
- ISO standard generally supported (with minor differences).
- *Basic* module system pretty compatible.
- A good number of commonly available features:
  - Constraints.
  - Multi-threading.
  - Tabling.
  - Coroutining.
  - ...

However,

- Interfaces and details often differ.
  Can mostly be bridged (c.f., Paolo Moura's work), but a real nuisance.
- Some features (e.g., Types/modes/verification, s(CASP), ...)
  still in few systems.

Summary (so far)

- Prolog systems have come a long way!
- ISO standard generally supported (with minor differences).
- *Basic* module system pretty compatible.
- A good number of commonly available features:
  - Constraints.
  - Multi-threading.
  - Tabling.
  - Coroutining.
  - ...

However,

- Interfaces and details often differ.
  Can mostly be bridged (c.f., Paolo Moura's work), but a real nuisance.
- Some features (e.g., Types/modes/verification, s(CASP), ...)
  still in few systems.

Summary (so far)

- Prolog systems have come a long way!
- ISO standard generally supported (with minor differences).
- *Basic* module system pretty compatible.
- A good number of commonly available features:
  - Constraints.
  - Multi-threading.
  - Tabling.
  - Coroutining.
  - ...

  However,
  - Interfaces and details often differ.
    Can mostly be bridged (c.f., Paolo Moura's work), but a real nuisance.
  - Some features (e.g., Types/modes/verification, s(CASP), ...)
    still in few systems.

Summary (so far)

- Prolog systems have come a long way!
- ISO standard generally supported (with minor differences).
- *Basic* module system pretty compatible.
- A good number of commonly available features:
  - Constraints.
  - Multi-threading.
  - Tabling.
  - Coroutining.
  - ...

  However,
  - Interfaces and details often differ.
    Can mostly be bridged (c.f., Paolo Moura's work), but a real nuisance.
  - Some features (e.g., Types/modes/verification, s(CASP), ...)
    still in few systems.

Influences on others

Influence in other languages within LP and its extensions

- Goedel, Mercury, Turbo-Prolog (static typing)
- λ-Prolog, Curry, Babel
- CP, GHC, Parlog, Erlang (committed choice)
- Datalog, ASP
- s(ASP) and s(CASP) (can also be seen as extensions)
- HyProlog, Co-inductive LP, ...
- Probabilistic LP
- LogTalk
- Picat
- CHR, CHRG
- ...

Influence beyond LP

- Theorem proving technology.
- Java (abstract machine, specification, ...).
- Erlang.
- Many embeddings in other languages.
- Many others: C++, many compilers, ...
- Analyzers and verifiers for other languages.
- ...

Further analysis of current status and outlook

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with “declarative” pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with “declarative” pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with “declarative” pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with “declarative” pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!
- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Threats to Prolog's future $\rightarrow$ and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

$\rightarrow$

- Many weaknesses already addressed by different systems. $\rightarrow$ continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.
- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.
  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.
- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.
  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.

- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.
  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.
- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.
  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.
- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.
  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.
- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.
  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.
- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.
  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.
- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.
  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.
- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.
  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.
- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.
  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.
- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.
  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.
- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.
  - ...

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
-
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
-
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
-
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
-
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
- Types and modes, and other properties.
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
- Types and modes, and other properties.
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
- Types and modes, and other properties.
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
- Types and modes, and other properties.
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
- Types and modes, and other properties.
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
- Types and modes, and other properties.
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
- Types and modes, and other properties.
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
- Types and modes, and other properties.
- Reactivity.
- ... (also, community infrastructure, see at the end).

Types, modes, and other properties

(Some perspectives from the Ciao Prolog system)

Dynamic vs. Static languages – the classic dilemma

Dynamic languages **(Prolog, Lisp/Scheme, Python, Javascript, ...)**

- Dynamic checking of basic types, modes, and some other properties:
  - `..., A is B+C, ...`
    `B` and `C` checked by `is/2` to be *instantiated* to `numexpr` at run time.
  - `..., arg(N,T,A), ...`
    `N` checked by `arg/3` to be `nat` & $\leq$ `arity(T)` ("array bounds").

→ Flexibility, compactness, rapid prototyping, scripting, ..., but
  - Most errors only detected at run time.
  - Need to use tags (*boxing* of data) to identify type and mode, store arity, etc.

Static languages **(ML, Haskell, Mercury, Gödel, ...)**

- Compiler statically checks *types*.

→ Safety guarantees (types), performance, scalability, ..., but
more rigid, limitations on language and provable properties.

- Note that some languages (e.g., C) are neither (even if still very useful!):
  no checking of, e.g., array bounds at compile time or run time...

Dynamic vs. Static languages – the classic dilemma

Dynamic languages **(Prolog, Lisp/Scheme, Python, Javascript, ...)**

- Dynamic checking of basic types, modes, and some other properties:
  - `..., A is B+C, ...`
    `B` and `C` checked by `is/2` to be *instantiated* to `numexpr` at run time.
  - `..., arg(N,T,A), ...`
    `N` checked by `arg/3` to be `nat` & $\leq$ `arity(T)` ("array bounds").

→ Flexibility, compactness, rapid prototyping, scripting, ..., but
  - Most errors only detected at run time.
  - Need to use tags (*boxing* of data) to identify type and mode, store arity, etc.

Static languages **(ML, Haskell, Mercury, Gödel, ...)**

- Compiler statically checks *types*.

→ Safety guarantees (types), performance, scalability, ..., but
more rigid, limitations on language and provable properties.

- Note that some languages (e.g., C) are neither (even if still very useful!):
  no checking of, e.g., array bounds at compile time or run time...

Dynamic vs. Static languages – the classic dilemma

Dynamic languages **(Prolog, Lisp/Scheme, Python, Javascript, ...)**

- Dynamic checking of basic types, modes, and some other properties:
  - `..., A is B+C, ...`
    `B` and `C` checked by `is/2` to be *instantiated* to `numexpr` at run time.
  - `..., arg(N,T,A), ...`
    `N` checked by `arg/3` to be `nat` & $\leq$ `arity(T)` ("array bounds").

→ Flexibility, compactness, rapid prototyping, scripting, ..., but
  - Most errors only detected at run time.
  - Need to use tags (*boxing* of data) to identify type and mode, store arity, etc.

Static languages **(ML, Haskell, Mercury, Gödel, ...)**

- Compiler statically checks *types*.

→ Safety guarantees (types), performance, scalability, ..., but more rigid, limitations on language and provable properties.

- Note that some languages (e.g., C) are neither (even if still very useful!): no checking of, e.g., array bounds at compile time or run time...

Dynamic vs. Static languages – the classic dilemma

Dynamic languages **(Prolog, Lisp/Scheme, Python, Javascript, ...)**

- Dynamic checking of basic types, modes, and some other properties:
  - `..., A is B+C, ...`
    `B` and `C` checked by `is/2` to be *instantiated* to `numexpr` at run time.
  - `..., arg(N,T,A), ...`
    `N` checked by `arg/3` to be `nat` & $\leq$ `arity(T)` ("array bounds").

→ Flexibility, compactness, rapid prototyping, scripting, ..., but
  - Most errors only detected at run time.
  - Need to use tags (*boxing* of data) to identify type and mode, store arity, etc.

Static languages **(ML, Haskell, Mercury, Gödel, ...)**

- Compiler statically checks *types*.

→ Safety guarantees (types), performance, scalability, ..., but
more rigid, limitations on language and provable properties.

- Note that some languages (e.g., C) are neither (even if still very useful!):
  no checking of, e.g., array bounds at compile time or run time...

Dynamic vs. Static languages – the classic dilemma

Dynamic languages **(Prolog, Lisp/Scheme, Python, Javascript, ...)**

- Dynamic checking of basic types, modes, and some other properties:
  - `..., A is B+C, ...`
    `B` and `C` checked by `is/2` to be *instantiated* to `numexpr` at run time.
  - `..., arg(N,T,A), ...`
    `N` checked by `arg/3` to be `nat` & $\leq$ `arity(T)` ("array bounds").

→ Flexibility, compactness, rapid prototyping, scripting, ..., but
  - Most errors only detected at run time.
  - Need to use tags (*boxing* of data) to identify type and mode, store arity, etc.

Static languages **(ML, Haskell, Mercury, Gödel, ...)**

- Compiler statically checks *types*.

→ Safety guarantees (types), performance, scalability, ..., but
more rigid, limitations on language and provable properties.

- Note that some languages (e.g., C) are neither (even if still very useful!):
  no checking of, e.g., array bounds at compile time or run time...

Dynamic vs. Static languages – the classic dilemma

Dynamic languages **(Prolog, Lisp/Scheme, Python, Javascript, ...)**

- Dynamic checking of basic types, modes, and some other properties:
  - `..., A is B+C, ...`
    `B` and `C` checked by `is/2` to be *instantiated* to `numexpr` at run time.
  - `..., arg(N,T,A), ...`
    `N` checked by `arg/3` to be `nat` & $\leq$ `arity(T)` ("array bounds").

→ Flexibility, compactness, rapid prototyping, scripting, ..., but
  - Most errors only detected at run time.
  - Need to use tags (*boxing* of data) to identify type and mode, store arity, etc.

Static languages **(ML, Haskell, Mercury, Gödel, ...)**

- Compiler statically checks *types*.

→ Safety guarantees (types), performance, scalability, ..., but
more rigid, limitations on language and provable properties.

- Note that some languages (e.g., C) are neither (even if still very useful!):
  no checking of, e.g., array bounds at compile time or run time...

Dynamic vs. Static languages – the classic dilemma

Dynamic languages **(Prolog, Lisp/Scheme, Python, Javascript, ...)**

- Dynamic checking of basic types, modes, and some other properties:
  - `..., A is B+C, ...`
    `B` and `C` checked by `is/2` to be *instantiated* to `numexpr` at run time.
  - `..., arg(N,T,A), ...`
    `N` checked by `arg/3` to be `nat` & $\leq$ `arity(T)` ("array bounds").

→ Flexibility, compactness, rapid prototyping, scripting, ..., but
  - Most errors only detected at run time.
  - Need to use tags (*boxing* of data) to identify type and mode, store arity, etc.

Static languages **(ML, Haskell, Mercury, Gödel, ...)**

- Compiler statically checks *types*.

→ Safety guarantees (types), performance, scalability, ..., but more rigid, limitations on language and provable properties.

- Note that some languages (e.g., C) are neither (even if still very useful!): no checking of, e.g., array bounds at compile time or run time...

Solving the Dynamic vs. Static Dilemma

The Ciao Approach (mid 90s's!):

1. *Assertions* can be used to express types, modes, and many other properties.
   - But voluntary: provided up front, gradually, or not at all.
2. Then, *advanced program analysis* (abstract interpretation) is used to:
   - Verify the assertions:
     - As much as possible at compile-time;
     - else, run-time tests generated.
   - Achieve high performance:
     - Eliminate run-time checks at compile time.
     - Unboxing, specialization, slicing, automatic parallelization, ...
3. Also, easily generate tests from assertions (this is (C)LP!).

- Provides the flexibility of dynamic languages, but with
- *guaranteed safety, reliability, and efficiency.*

- Quite popular nowadays: gradual typing, Racket, liquid Haskell, etc.

Solving the Dynamic vs. Static Dilemma

The Ciao Approach (mid 90s's!):

1. *Assertions* can be used to express types, modes, and many other properties.
   - But voluntary: provided up front, gradually, or not at all.
2. Then, *advanced program analysis* (abstract interpretation) is used to:
   - Verify the assertions:
     - As much as possible at compile-time;
     - else, run-time tests generated.
   - Achieve high performance:
     - Eliminate run-time checks at compile time.
     - Unboxing, specialization, slicing, automatic parallelization, ...
3. Also, easily generate tests from assertions (this is (C)LP!).

- Provides the flexibility of dynamic languages, but with
- *guaranteed safety, reliability, and efficiency.*

- Quite popular nowadays: gradual typing, Racket, liquid Haskell, etc.

Solving the Dynamic vs. Static Dilemma

The Ciao Approach (mid 90s's!):

1. *Assertions* can be used to express types, modes, and many other properties.
   - But voluntary: provided up front, gradually, or not at all.
2. Then, *advanced program analysis* (abstract interpretation) is used to:
   - Verify the assertions:
     - As much as possible at compile-time;
     - else, run-time tests generated.
   - Achieve high performance:
     - Eliminate run-time checks at compile time.
     - Unboxing, specialization, slicing, automatic parallelization, ...
3. Also, easily generate tests from assertions (this is (C)LP!).

- Provides the flexibility of dynamic languages, but with
- *guaranteed safety, reliability, and efficiency.*

- Quite popular nowadays: gradual typing, Racket, liquid Haskell, etc.

Solving the Dynamic vs. Static Dilemma

The Ciao Approach (mid 90s's!):

1. *Assertions* can be used to express types, modes, and many other properties.
   - But voluntary: provided up front, gradually, or not at all.
2. Then, *advanced program analysis* (abstract interpretation) is used to:
   - Verify the assertions:
     - As much as possible at compile-time;
     - else, run-time tests generated.
   - Achieve high performance:
     - Eliminate run-time checks at compile time.
     - Unboxing, specialization, slicing, automatic parallelization, ...
3. Also, easily generate tests from assertions (this is (C)LP!).

- Provides the flexibility of dynamic languages, but with
- *guaranteed safety, reliability, and efficiency.*

- Quite popular nowadays: gradual typing, Racket, liquid Haskell, etc.

Solving the Dynamic vs. Static Dilemma

The Ciao Approach (mid 90s's!):

1. *Assertions* can be used to express types, modes, and many other properties.
   - But voluntary: provided up front, gradually, or not at all.
2. Then, *advanced program analysis* (abstract interpretation) is used to:
   - Verify the assertions:
     - As much as possible at compile-time;
     - else, run-time tests generated.
   - Achieve high performance:
     - Eliminate run-time checks at compile time.
     - Unboxing, specialization, slicing, automatic parallelization, ...
3. Also, easily generate tests from assertions (this is (C)LP!).

- Provides the flexibility of dynamic languages, but with
- *guaranteed safety, reliability, and efficiency.*

- Quite popular nowadays: gradual typing, Racket, liquid Haskell, etc.

Solving the Dynamic vs. Static Dilemma

The Ciao Approach (mid 90s's!):

1. *Assertions* can be used to express types, modes, and many other properties.
   - But voluntary: provided up front, gradually, or not at all.
2. Then, *advanced program analysis* (abstract interpretation) is used to:
   - Verify the assertions:
     - As much as possible at compile-time;
     - else, run-time tests generated.
   - Achieve high performance:
     - Eliminate run-time checks at compile time.
     - Unboxing, specialization, slicing, automatic parallelization, ...
3. Also, easily generate tests from assertions (this is (C)LP!).

- Provides the flexibility of dynamic languages, but with
- *guaranteed safety, reliability, and efficiency.*

- Quite popular nowadays: gradual typing, Racket, liquid Haskell, etc.

Solving the Dynamic vs. Static Dilemma

The Ciao Approach (mid 90s's!):

1. *Assertions* can be used to express types, modes, and many other properties.
   - But voluntary: provided up front, gradually, or not at all.
2. Then, *advanced program analysis* (abstract interpretation) is used to:
   - Verify the assertions:
     - As much as possible at compile-time;
     - else, run-time tests generated.
   - Achieve high performance:
     - Eliminate run-time checks at compile time.
     - Unboxing, specialization, slicing, automatic parallelization, ...
3. Also, easily generate tests from assertions (this is (C)LP!).

- Provides the flexibility of dynamic languages, but with
- *guaranteed safety, reliability, and efficiency.*

- Quite popular nowadays: gradual typing, Racket, liquid Haskell, etc.

Solving the Dynamic vs. Static Dilemma

The Ciao Approach (mid 90s's!):

1. *Assertions* can be used to express types, modes, and many other properties.
   - But voluntary: provided up front, gradually, or not at all.
2. Then, *advanced program analysis* (abstract interpretation) is used to:
   - Verify the assertions:
     - As much as possible at compile-time;
     - else, run-time tests generated.
   - Achieve high performance:
     - Eliminate run-time checks at compile time.
     - Unboxing, specialization, slicing, automatic parallelization, ...
3. Also, easily generate tests from assertions (this is (C)LP!).

- Provides the flexibility of dynamic languages, but with
- *guaranteed safety, reliability, and efficiency.*

- Quite popular nowadays: gradual typing, Racket, liquid Haskell, etc.

Solving the Dynamic vs. Static Dilemma

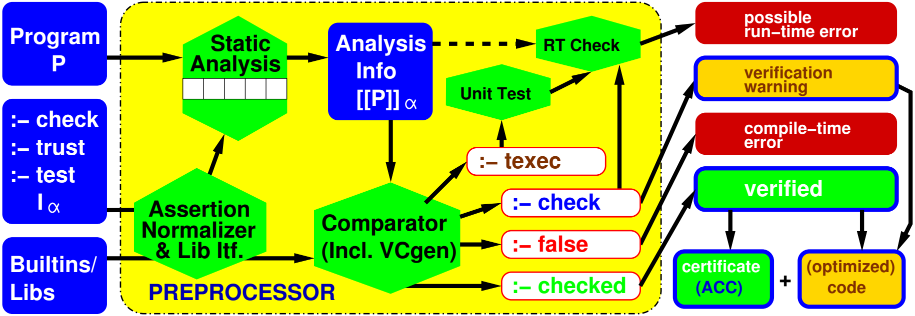
The Ciao Approach (mid 90s's!):

1. *Assertions* can be used to express types, modes, and many other properties.
   - But voluntary: provided up front, gradually, or not at all.
2. Then, *advanced program analysis* (abstract interpretation) is used to:
   - Verify the assertions:
     - As much as possible at compile-time;
     - else, run-time tests generated.
   - Achieve high performance:
     - Eliminate run-time checks at compile time.
     - Unboxing, specialization, slicing, automatic parallelization, ...
3. Also, easily generate tests from assertions (this is (C)LP!).

- Provides the flexibility of dynamic languages, but with
- *guaranteed safety, reliability, and efficiency.*

- Quite popular nowadays: gradual typing, Racket, liquid Haskell, etc.

The Ciao Integrated Approach to Specification, Debugging, Verification, Testing, and Optimization



Discussion: Comparison with Classical Types

"Traditional" Types	Ciao Assertion-based Model
"Properties" limited by decidability	Much more general property language
May need to limit prog. lang.	No need to limit prog. lang.
"Untypable" programs rejected	Run-time checks introduced
(Almost) Decidable	Decidable + Undecidable (approximated)
Expressed in a different language	Expressed in the source language
Types must be defined	Types can be defined or inferred
Assertions are only of type "check"	"check", "trust", ...
Type signatures & assertions different	Type signatures *are* assertions

- But quite popular now: gradual typing, Racket, liquid Haskell, etc.
- Some key issues:
  Safe / Sound approximation — *Suitable assertion language*
  Abstract Interpretation — *Powerful abstract domains*
- Works best when properties and assertions can be expressed in the source language (i.e., source lang. supports *predicates*, *constraints*).

Discussion: Comparison with Classical Types

"Traditional" Types	Ciao Assertion-based Model
"Properties" limited by decidability	Much more general property language
May need to limit prog. lang.	No need to limit prog. lang.
"Untypable" programs rejected	Run-time checks introduced
(Almost) Decidable	Decidable + Undecidable (approximated)
Expressed in a different language	Expressed in the source language
Types must be defined	Types can be defined or inferred
Assertions are only of type "check"	"check", "trust", ...
Type signatures & assertions different	Type signatures *are* assertions

- But quite popular now: gradual typing, Racket, liquid Haskell, etc.

- Some key issues:
  Safe / Sound approximation — *Suitable assertion language*
  Abstract Interpretation — *Powerful abstract domains*

- Works best when properties and assertions can be expressed in the source language (i.e., source lang. supports *predicates*, *constraints*).

Discussion: Comparison with Classical Types

"Traditional" Types	**Ciao Assertion-based Model**
"Properties" limited by decidability	Much more general property language
May need to limit prog. lang.	No need to limit prog. lang.
"Untypable" programs rejected	Run-time checks introduced
(Almost) Decidable	Decidable + Undecidable (approximated)
Expressed in a different language	Expressed in the source language
Types must be defined	Types can be defined or inferred
Assertions are only of type "check"	"check", "trust", ...
Type signatures & assertions different	Type signatures *are* assertions

- But quite popular now: gradual typing, Racket, liquid Haskell, etc.

- Some key issues:
  Safe / Sound approximation — *Suitable assertion language*
  Abstract Interpretation — *Powerful abstract domains*

- Works best when properties and assertions can be expressed in the source language (i.e., source lang. supports *predicates*, *constraints*).

Demo! (See slides at the end.)

Teaching (and preaching) Prolog

On teaching (and preaching) Prolog

- Prolog / LP / CLP *must* be taught in CS programs,
  - A CS graduate is simply not complete without knowledge of Prolog.

  (and maybe also in other majors and maybe in schools –cf. Prolog Year?)

- But it has to be done right!
  - The standard 'programming paradigms' approach can be counter-productive.
  - Simply cannot be done in a couple of weeks emulating Prolog in Scheme.
    - What to do if that is the only slot available?

- On the way *dispel unfounded myths* about the language, and show how many of the shortcomings of early Prologs have been *addressed over the years*.

On teaching (and preaching) Prolog

- Prolog / LP / CLP *must* be taught in CS programs,
  - A CS graduate is simply not complete without knowledge of Prolog.

  (and maybe also in other majors and maybe in schools –cf. Prolog Year?)

- But it has to be done right!
  - The standard 'programming paradigms' approach can be counter-productive.
  - Simply cannot be done in a couple of weeks emulating Prolog in Scheme.
    - What to do if that is the only slot available?

- On the way *dispel unfounded myths* about the language, and show how many of the shortcomings of early Prologs have been *addressed over the years*.

On teaching (and preaching) Prolog

- Prolog / LP / CLP *must* be taught in CS programs,
  - A CS graduate is simply not complete without knowledge of Prolog.

  (and maybe also in other majors and maybe in schools –cf. Prolog Year?)

- But it has to be done right!
  - The standard 'programming paradigms' approach can be counter-productive.
  - Simply cannot be done in a couple of weeks emulating Prolog in Scheme.
    - What to do if that is the only slot available?

- On the way *dispel unfounded myths* about the language, and show how many of the shortcomings of early Prologs have been *addressed over the years*.

On teaching (and preaching) Prolog

- "Prolog gets into infinite loops."

  This is true –in fact, of any programming language or proof system.
  However, it is likely to discourage beginners if not explained well:

  - Use a system that can *alternatively and selectively* run in breadth-first, iterative deepening, tabling, etc.
  - Start by running all predicates, e.g., breadth-first – everything works!
  - Then, explain the shape of the tree (solutions at finite depth, possible infinite failures, etc.), and thus why breadth-first works, and why depth-first sometimes may not.
  - Do relate it to the *halting problem*: no-one (Prolog, logic, nor other Turing-complete prog. language) can solve that (but tabling helps).
  - Discuss advantages and disadvantages of search rules (time, memory). Motivate the choices made for Prolog benchmarking actual executions.

On teaching (and preaching) Prolog

- "Prolog gets into infinite loops."

  This is true –in fact, of any programming language or proof system. However, it is likely to discourage beginners if not explained well:

  - Use a system that can *alternatively and selectively* run in breadth-first, iterative deepening, tabling, etc.
  - Start by running all predicates, e.g., breadth-first – everything works!
  - Then, explain the shape of the tree (solutions at finite depth, possible infinite failures, etc.), and thus why breadth-first works, and why depth-first sometimes may not.
  - Do relate it to the *halting problem*: no-one (Prolog, logic, nor other Turing-complete prog. language) can solve that (but tabling helps).
  - Discuss advantages and disadvantages of search rules (time, memory). Motivate the choices made for Prolog benchmarking actual executions.

On teaching (and preaching) Prolog

- "Prolog gets into infinite loops."

  This is true –in fact, of any programming language or proof system.
  However, it is likely to discourage beginners if not explained well:

  - Use a system that can *alternatively and selectively* run in breadth-first, iterative deepening, tabling, etc.
  - Start by running all predicates, e.g., breadth-first – everything works!
  - Then, explain the shape of the tree (solutions at finite depth, possible infinite failures, etc.), and thus why breadth-first works, and why depth-first sometimes may not.
  - Do relate it to the *halting problem*: no-one (Prolog, logic, nor other Turing-complete prog. language) can solve that (but tabling helps).
  - Discuss advantages and disadvantages of search rules (time, memory). Motivate the choices made for Prolog benchmarking actual executions.

On teaching (and preaching) Prolog

- "Prolog gets into infinite loops."

  This is true –in fact, of any programming language or proof system. However, it is likely to discourage beginners if not explained well:

  - Use a system that can *alternatively and selectively* run in breadth-first, iterative deepening, tabling, etc.
  - Start by running all predicates, e.g., breadth-first – everything works!
  - Then, explain the shape of the tree (solutions at finite depth, possible infinite failures, etc.), and thus why breadth-first works, and why depth-first sometimes may not.
  - Do relate it to the *halting problem*: no-one (Prolog, logic, nor other Turing-complete prog. language) can solve that (but tabling helps).
  - Discuss advantages and disadvantages of search rules (time, memory). Motivate the choices made for Prolog benchmarking actual executions.

On teaching (and preaching) Prolog

- "Prolog gets into infinite loops."

  This is true –in fact, of any programming language or proof system.
  However, it is likely to discourage beginners if not explained well:

  - Use a system that can *alternatively and selectively* run in breadth-first, iterative deepening, tabling, etc.
  - Start by running all predicates, e.g., breadth-first – everything works!
  - Then, explain the shape of the tree (solutions at finite depth, possible infinite failures, etc.), and thus why breadth-first works, and why depth-first sometimes may not.
  - Do relate it to the *halting problem*: no-one (Prolog, logic, nor other Turing-complete prog. language) can solve that (but tabling helps).
  - Discuss advantages and disadvantages of search rules (time, memory). Motivate the choices made for Prolog benchmarking actual executions.

On teaching (and preaching) Prolog

- "Prolog gets into infinite loops."

  This is true –in fact, of any programming language or proof system.
  However, it is likely to discourage beginners if not explained well:

  - Use a system that can *alternatively and selectively* run in breadth-first, iterative deepening, tabling, etc.
  - Start by running all predicates, e.g., breadth-first – everything works!
  - Then, explain the shape of the tree (solutions at finite depth, possible infinite failures, etc.), and thus why breadth-first works, and why depth-first sometimes may not.
  - Do relate it to the *halting problem*: no-one (Prolog, logic, nor other Turing-complete prog. language) can solve that (but tabling helps).
  - Discuss advantages and disadvantages of search rules (time, memory). Motivate the choices made for Prolog benchmarking actual executions.

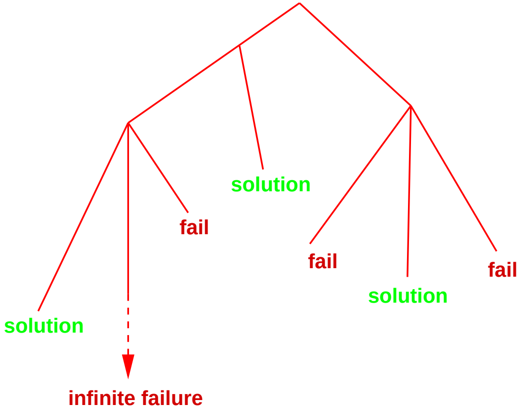
On teaching (and preaching) Prolog

- "Prolog gets into infinite loops."

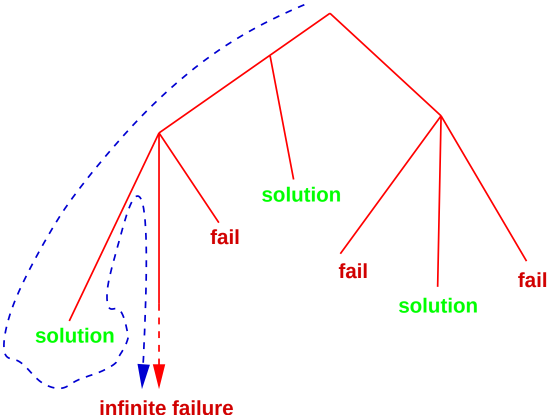
  This is true –in fact, of any programming language or proof system.
  However, it is likely to discourage beginners if not explained well:

  - Use a system that can *alternatively and selectively* run in breadth-first, iterative deepening, tabling, etc.
  - Start by running all predicates, e.g., breadth-first – everything works!
  - Then, explain the shape of the tree (solutions at finite depth, possible infinite failures, etc.), and thus why breadth-first works, and why depth-first sometimes may not.
  - Do relate it to the *halting problem*: no-one (Prolog, logic, nor other Turing-complete prog. language) can solve that (but tabling helps).
  - Discuss advantages and disadvantages of search rules (time, memory). Motivate the choices made for Prolog benchmarking actual executions.

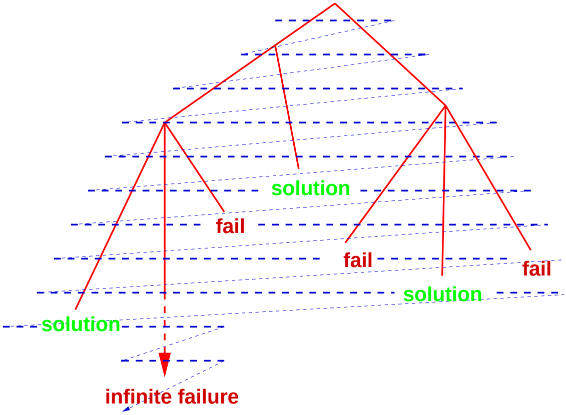
Characterization of the search tree



Depth-First Search



Breadth-First Search



On teaching (and preaching) Prolog

- "Arithmetic is not reversible."
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- "There is no occur check."
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- "Prolog is not pure (cut, assert, etc.)"
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching (and preaching) Prolog

- "Arithmetic is not reversible."
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- "There is no occur check."
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- "Prolog is not pure (cut, assert, etc.)"
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching (and preaching) Prolog

- "Arithmetic is not reversible."
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!

- "There is no occur check."
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).

- "Prolog is not pure (cut, assert, etc.)"
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching (and preaching) Prolog

- "Arithmetic is not reversible."
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- "There is no occur check."
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- "Prolog is not pure (cut, assert, etc.)"
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching (and preaching) Prolog

- "Arithmetic is not reversible."
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- "There is no occur check."
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- "Prolog is not pure (cut, assert, etc.)"
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching (and preaching) Prolog

- "Arithmetic is not reversible."
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- "There is no occur check."
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- "Prolog is not pure (cut, assert, etc.)"
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching (and preaching) Prolog

- "Arithmetic is not reversible."
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- "There is no occur check."
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- "Prolog is not pure (cut, assert, etc.)"
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching (and preaching) Prolog

- "Arithmetic is not reversible."
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- "There is no occur check."
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- "Prolog is not pure (cut, assert, etc.)"
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching (and preaching) Prolog

- "Arithmetic is not reversible."
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- "There is no occur check."
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- "Prolog is not pure (cut, assert, etc.)"
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching (and preaching) Prolog

- "Arithmetic is not reversible."
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- "There is no occur check."
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- "Prolog is not pure (cut, assert, etc.)"
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching (and preaching) Prolog

- "Arithmetic is not reversible."
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- "There is no occur check."
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- "Prolog is not pure (cut, assert, etc.)"
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching (and preaching) Prolog

- "Arithmetic is not reversible."
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- "There is no occur check."
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- "Prolog is not pure (cut, assert, etc.)"
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching (and preaching) Prolog

- "Arithmetic is not reversible."
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- "There is no occur check."
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- "Prolog is not pure (cut, assert, etc.)"
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching (and preaching) Prolog

- "Arithmetic is not reversible."
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- "There is no occur check."
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- "Prolog is not pure (cut, assert, etc.)"
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching (and preaching) Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- "Prolog is a strange language."
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- "Prolog has no applications / interest / nobody uses it."
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- "The Fifth Generation failed!" Not true...
  and it did not use Prolog or "real LP" anyway!
  They used in fact "something like Erlang"
  (probably why it was not as successful as it could have been.)

On teaching (and preaching) Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- "Prolog is a strange language."
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- "Prolog has no applications / interest / nobody uses it."
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- "The Fifth Generation failed!" Not true...
  and it did not use Prolog or "real LP" anyway!
  They used in fact "something like Erlang"
  (probably why it was not as successful as it could have been.)

On teaching (and preaching) Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- "Prolog is a strange language."
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- "Prolog has no applications / interest / nobody uses it."
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- "The Fifth Generation failed!" Not true...
  and it did not use Prolog or "real LP" anyway!
  They used in fact "something like Erlang"
  (probably why it was not as successful as it could have been.)

On teaching (and preaching) Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- "Prolog is a strange language."
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- "Prolog has no applications / interest / nobody uses it."
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- "The Fifth Generation failed!" Not true...
  and it did not use Prolog or "real LP" anyway!
  They used in fact "something like Erlang"
  (probably why it was not as successful as it could have been.)

On teaching (and preaching) Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- "Prolog is a strange language."
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- "Prolog has no applications / interest / nobody uses it."
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- "The Fifth Generation failed!" Not true...
  and it did not use Prolog or "real LP" anyway!
  They used in fact "something like Erlang"
  (probably why it was not as successful as it could have been.)

On teaching (and preaching) Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- "Prolog is a strange language."
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- "Prolog has no applications / interest / nobody uses it."
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- "The Fifth Generation failed!" Not true...
  and it did not use Prolog or "real LP" anyway!
  They used in fact "something like Erlang"
  (probably why it was not as successful as it could have been.)

On teaching (and preaching) Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- "Prolog is a strange language."
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- "Prolog has no applications / interest / nobody uses it."
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- "The Fifth Generation failed!" Not true...
  and it did not use Prolog or "real LP" anyway!
  They used in fact "something like Erlang"
  (probably why it was not as successful as it could have been.)

On teaching (and preaching) Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- "Prolog is a strange language."
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- "Prolog has no applications / interest / nobody uses it."
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- "The Fifth Generation failed!" Not true...
  and it did not use Prolog or "real LP" anyway!
  They used in fact "something like Erlang"
  (probably why it was not as successful as it could have been.)

On teaching (and preaching) Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- "Prolog is a strange language."
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- "Prolog has no applications / interest / nobody uses it."
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- "The Fifth Generation failed!" Not true...
  and it did not use Prolog or "real LP" anyway!
  They used in fact "something like Erlang"
  (probably why it was not as successful as it could have been.)

On teaching (and preaching) Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- "Prolog is a strange language."
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- "Prolog has no applications / interest / nobody uses it."
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- "The Fifth Generation failed!" Not true...
  and it did not use Prolog or "real LP" anyway!
  They used in fact "something like Erlang"
  (probably why it was not as successful as it could have been.)

On teaching (and preaching) Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- "Prolog is a strange language."
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- "Prolog has no applications / interest / nobody uses it."
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- "The Fifth Generation failed!" Not true...
  and it did not use Prolog or "real LP" anyway!
  They used in fact "something like Erlang"
  (probably why it was not as successful as it could have been.)

Personal Sequential Inference –PSI– machine (Prolog machine) in FGCS ICOT's basement (the large refrigerator-size box on the right).

On teaching (and preaching) Prolog

- Do show the beauty:
  - Explain "Green's dream," discuss for what logics we have effective deduction procedures, justify the choice of FO and semi-decidability, SLD-resolution → classical LP (Kowalski/Colmerauer).
  - Show how logic programs are both logical theories (with declarative meaning) and procedural programs that can be debugged, followed step by step, etc.
    - An operational (in addition to declarative) semantics is a requirement in the language (vs., e.g., Goedel) and we do need to teach it.
    - Otherwise not a programming language, just specification/KR – Prolog is both.
    - How otherwise to reason about complexity, memory consumption, etc.? To say that these things don't matter does not make sense in PL.
  - Show with examples (and benchmarking them) how you can go from executable specifications to efficient algorithms gradually, and as needed.
  - Show how unification is also a device for *constructing and matching complex data structures with (declarative) pointers*. Show it in the top level, giving "the data structures class."
  - Do use types (and properties in general): define them as predicates, show them used to check if something is in the type (dynamic checking), or "run backwards" to generate the "inhabitants"; property-based testing for free!

On teaching (and preaching) Prolog

- Do show the beauty:
  - Explain "Green's dream," discuss for what logics we have effective deduction procedures, justify the choice of FO and semi-decidability, SLD-resolution → classical LP (Kowalski/Colmerauer).
  - Show how logic programs are both logical theories (with declarative meaning) and procedural programs that can be debugged, followed step by step, etc.
    - An operational (in addition to declarative) semantics is a requirement in the language (vs., e.g., Goedel) and we do need to teach it.
    - Otherwise not a programming language, just specification/KR – Prolog is both.
    - How otherwise to reason about complexity, memory consumption, etc.? To say that these things don't matter does not make sense in PL.
  - Show with examples (and benchmarking them) how you can go from executable specifications to efficient algorithms gradually, and as needed.
  - Show how unification is also a device for *constructing and matching complex data structures with (declarative) pointers*. Show it in the top level, giving "the data structures class."
  - Do use types (and properties in general): define them as predicates, show them used to check if something is in the type (dynamic checking), or "run backwards" to generate the "inhabitants"; property-based testing for free!

On teaching (and preaching) Prolog

- Do show the beauty:
  - Explain "Green's dream," discuss for what logics we have effective deduction procedures, justify the choice of FO and semi-decidability, SLD-resolution $\rightarrow$ classical LP (Kowalski/Colmerauer).
  - Show how logic programs are both logical theories (with declarative meaning) and procedural programs that can be debugged, followed step by step, etc.
    - An operational (in addition to declarative) semantics is a requirement in the language (vs., e.g., Goedel) and we do need to teach it.
    - Otherwise not a programming language, just specification/KR – Prolog is both.
    - How otherwise to reason about complexity, memory consumption, etc.? To say that these things don't matter does not make sense in PL.
  - Show with examples (and benchmarking them) how you can go from executable specifications to efficient algorithms gradually, and as needed.
  - Show how unification is also a device for *constructing and matching complex data structures with (declarative) pointers*. Show it in the top level, giving "the data structures class."
  - Do use types (and properties in general): define them as predicates, show them used to check if something is in the type (dynamic checking), or "run backwards" to generate the "inhabitants"; property-based testing for free!

On teaching (and preaching) Prolog

- Do show the beauty:
  - Explain "Green's dream," discuss for what logics we have effective deduction procedures, justify the choice of FO and semi-decidability, SLD-resolution → classical LP (Kowalski/Colmerauer).
  - Show how logic programs are both logical theories (with declarative meaning) and procedural programs that can be debugged, followed step by step, etc.
    - An operational (in addition to declarative) semantics is a requirement in the language (vs., e.g., Goedel) and we do need to teach it.
    - Otherwise not a programming language, just specification/KR – Prolog is both.
    - How otherwise to reason about complexity, memory consumption, etc.? To say that these things don't matter does not make sense in PL.
  - Show with examples (and benchmarking them) how you can go from executable specifications to efficient algorithms gradually, and as needed.
  - Show how unification is also a device for *constructing and matching complex data structures with (declarative) pointers*. Show it in the top level, giving "the data structures class."
  - Do use types (and properties in general): define them as predicates, show them used to check if something is in the type (dynamic checking), or "run backwards" to generate the "inhabitants"; property-based testing for free!

On teaching (and preaching) Prolog

- Do show the beauty:
  - Explain "Green's dream," discuss for what logics we have effective deduction procedures, justify the choice of FO and semi-decidability, SLD-resolution $\rightarrow$ classical LP (Kowalski/Colmerauer).
  - Show how logic programs are both logical theories (with declarative meaning) and procedural programs that can be debugged, followed step by step, etc.
    - An operational (in addition to declarative) semantics is a requirement in the language (vs., e.g., Goedel) and we do need to teach it.
    - Otherwise not a programming language, just specification/KR – Prolog is both.
    - How otherwise to reason about complexity, memory consumption, etc.? To say that these things don't matter does not make sense in PL.
  - Show with examples (and benchmarking them) how you can go from executable specifications to efficient algorithms gradually, and as needed.
  - Show how unification is also a device for *constructing and matching complex data structures with (declarative) pointers*. Show it in the top level, giving "the data structures class."
  - Do use types (and properties in general): define them as predicates, show them used to check if something is in the type (dynamic checking), or "run backwards" to generate the "inhabitants"; property-based testing for free!

On teaching (and preaching) Prolog

- Do show the beauty:
  - Explain "Green's dream," discuss for what logics we have effective deduction procedures, justify the choice of FO and semi-decidability, SLD-resolution → classical LP (Kowalski/Colmerauer).
  - Show how logic programs are both logical theories (with declarative meaning) and procedural programs that can be debugged, followed step by step, etc.
    - An operational (in addition to declarative) semantics is a requirement in the language (vs., e.g., Goedel) and we do need to teach it.
    - Otherwise not a programming language, just specification/KR – Prolog is both.
    - How otherwise to reason about complexity, memory consumption, etc.? To say that these things don't matter does not make sense in PL.
  - Show with examples (and benchmarking them) how you can go from executable specifications to efficient algorithms gradually, and as needed.
  - Show how unification is also a device for *constructing and matching complex data structures with (declarative) pointers*. Show it in the top level, giving "the data structures class."
  - Do use types (and properties in general): define them as predicates, show them used to check if something is in the type (dynamic checking), or "run backwards" to generate the "inhabitants"; property-based testing for free!

On teaching (and preaching) Prolog

- Do show the beauty:
  - Explain "Green's dream," discuss for what logics we have effective deduction procedures, justify the choice of FO and semi-decidability, SLD-resolution → classical LP (Kowalski/Colmerauer).
  - Show how logic programs are both logical theories (with declarative meaning) and procedural programs that can be debugged, followed step by step, etc.
    - An operational (in addition to declarative) semantics is a requirement in the language (vs., e.g., Goedel) and we do need to teach it.
    - Otherwise not a programming language, just specification/KR – Prolog is both.
    - How otherwise to reason about complexity, memory consumption, etc.? To say that these things don't matter does not make sense in PL.
  - Show with examples (and benchmarking them) how you can go from executable specifications to efficient algorithms gradually, and as needed.
  - Show how unification is also a device for *constructing and matching complex data structures with (declarative) pointers*. Show it in the top level, giving "the data structures class."
  - Do use types (and properties in general): define them as predicates, show them used to check if something is in the type (dynamic checking), or "run backwards" to generate the "inhabitants"; property-based testing for free!

On teaching (and preaching) Prolog

- Do show the beauty:
  - Explain "Green's dream," discuss for what logics we have effective deduction procedures, justify the choice of FO and semi-decidability, SLD-resolution → classical LP (Kowalski/Colmerauer).
  - Show how logic programs are both logical theories (with declarative meaning) and procedural programs that can be debugged, followed step by step, etc.
    - An operational (in addition to declarative) semantics is a requirement in the language (vs., e.g., Goedel) and we do need to teach it.
    - Otherwise not a programming language, just specification/KR – Prolog is both.
    - How otherwise to reason about complexity, memory consumption, etc.? To say that these things don't matter does not make sense in PL.
  - Show with examples (and benchmarking them) how you can go from executable specifications to efficient algorithms gradually, and as needed.
  - Show how unification is also a device for *constructing and matching complex data structures with (declarative) pointers*. Show it in the top level, giving "the data structures class."
  - Do use types (and properties in general): define them as predicates, show them used to check if something is in the type (dynamic checking), or "run backwards" to generate the "inhabitants"; property-based testing for free!

On teaching (and preaching) Prolog

- Do show the beauty:
  - Explain "Green's dream," discuss for what logics we have effective deduction procedures, justify the choice of FO and semi-decidability, SLD-resolution → classical LP (Kowalski/Colmerauer).
  - Show how logic programs are both logical theories (with declarative meaning) and procedural programs that can be debugged, followed step by step, etc.
    - An operational (in addition to declarative) semantics is a requirement in the language (vs., e.g., Goedel) and we do need to teach it.
    - Otherwise not a programming language, just specification/KR – Prolog is both.
    - How otherwise to reason about complexity, memory consumption, etc.? To say that these things don't matter does not make sense in PL.
  - Show with examples (and benchmarking them) how you can go from executable specifications to efficient algorithms gradually, and as needed.
  - Show how unification is also a device for *constructing and matching complex data structures with (declarative) pointers*. Show it in the top level, giving "the data structures class."
  - Do use types (and properties in general): define them as predicates, show them used to check if something is in the type (dynamic checking), or "run backwards" to generate the "inhabitants"; property-based testing for free!

On teaching (and preaching) Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / "real" use.
    Show serious, competitive language.
  - Playgrounds and notebooks
    (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
    Very useful for executable examples in manuals and tutorials.
- Ideally the system should allow covering:
  - pure LP (with several search rules, tabling),
  - ISO-Prolog,
  - higher-order (and functional programming),
  - constraints,
  - ASP/s(CASP),
  - etc.

On teaching (and preaching) Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / “real” use.
    Show serious, competitive language.
  - Playgrounds and notebooks
    (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
    Very useful for executable examples in manuals and tutorials.
- Ideally the system should allow covering:
  - pure LP (with several search rules, tabling),
  - ISO-Prolog,
  - higher-order (and functional programming),
  - constraints,
  - ASP/s(CASP),
  - etc.

On teaching (and preaching) Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / "real" use.
    Show serious, competitive language.
  - Playgrounds and notebooks
    (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
    Very useful for executable examples in manuals and tutorials.
- Ideally the system should allow covering:
  - pure LP (with several search rules, tabling),
  - ISO-Prolog,
  - higher-order (and functional programming),
  - constraints,
  - ASP/s(CASP),
  - etc.

On teaching (and preaching) Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / “real” use.
    Show serious, competitive language.
  - Playgrounds and notebooks
    (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
    Very useful for executable examples in manuals and tutorials.
- Ideally the system should allow covering:
  - pure LP (with several search rules, tabling),
  - ISO-Prolog,
  - higher-order (and functional programming),
  - constraints,
  - ASP/s(CASP),
  - etc.

On teaching (and preaching) Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / “real” use.
    Show serious, competitive language.
  - Playgrounds and notebooks
    (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
    Very useful for executable examples in manuals and tutorials.
- Ideally the system should allow covering:
  - pure LP (with several search rules, tabling),
  - ISO-Prolog,
  - higher-order (and functional programming),
  - constraints,
  - ASP/s(CASP),
  - etc.

On teaching (and preaching) Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / “real” use.
    Show serious, competitive language.
  - Playgrounds and notebooks
    (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
    Very useful for executable examples in manuals and tutorials.
- Ideally the system should allow covering:
  - pure LP (with several search rules, tabling),
  - ISO-Prolog,
  - higher-order (and functional programming),
  - constraints,
  - ASP/s(CASP),
  - etc.

On teaching (and preaching) Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / “real” use.
    Show serious, competitive language.
  - Playgrounds and notebooks
    (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
    Very useful for executable examples in manuals and tutorials.
- Ideally the system should allow covering:
  - pure LP (with several search rules, tabling),
  - ISO-Prolog,
  - higher-order (and functional programming),
  - constraints,
  - ASP/s(CASP),
  - etc.

On teaching (and preaching) Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / "real" use.
    Show serious, competitive language.
  - Playgrounds and notebooks
    (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
    Very useful for executable examples in manuals and tutorials.
- Ideally the system should allow covering:
  - pure LP (with several search rules, tabling),
  - ISO-Prolog,
  - higher-order (and functional programming),
  - constraints,
  - ASP/s(CASP),
  - etc.

On teaching (and preaching) Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / “real” use.
    Show serious, competitive language.
  - Playgrounds and notebooks
    (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
    Very useful for executable examples in manuals and tutorials.
- Ideally the system should allow covering:
  - pure LP (with several search rules, tabling),
  - ISO-Prolog,
  - higher-order (and functional programming),
  - constraints,
  - ASP/s(CASP),
  - etc.

On teaching (and preaching) Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / "real" use.
    Show serious, competitive language.
  - Playgrounds and notebooks
    (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
    Very useful for executable examples in manuals and tutorials.
- Ideally the system should allow covering:
  - pure LP (with several search rules, tabling),
  - ISO-Prolog,
  - higher-order (and functional programming),
  - constraints,
  - ASP/s(CASP),
  - etc.

On teaching (and preaching) Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / "real" use.
    Show serious, competitive language.
  - Playgrounds and notebooks
    (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
    Very useful for executable examples in manuals and tutorials.
- Ideally the system should allow covering:
  - pure LP (with several search rules, tabling),
  - ISO-Prolog,
  - higher-order (and functional programming),
  - constraints,
  - ASP/s(CASP),
  - etc.

On teaching (and preaching) Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / “real” use.
    Show serious, competitive language.
  - Playgrounds and notebooks
    (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
    Very useful for executable examples in manuals and tutorials.
- Ideally the system should allow covering:
  - pure LP (with several search rules, tabling),
  - ISO-Prolog,
  - higher-order (and functional programming),
  - constraints,
  - ASP/s(CASP),
  - etc.

On teaching (and preaching) Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / “real” use.
    Show serious, competitive language.
  - Playgrounds and notebooks
    (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
    Very useful for executable examples in manuals and tutorials.
- Ideally the system should allow covering:
  - pure LP (with several search rules, tabling),
  - ISO-Prolog,
  - higher-order (and functional programming),
  - constraints,
  - ASP/s(CASP),
  - etc.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Demo slides for the part on:

Types, modes, and other properties

(Some perspectives from the Ciao Prolog system)

Example: qsort

Ciao warns that it cannot verify that the call to =</2 will not generate a run-time error (assertion is in library!):

```
:- module(_,[qsort/2],[assertions,nativeprops,.(nmodes)]).

qsort([], []).
qsort([First|Rest],Result) :-
    partition(Rest,First,Sm,Lg),
    qsort(Sm,SmS),
    qsort(Lg,LgS),
    append(SmS,[First|LgS],Result).

partition([],_,[],[]).
partition([X|Y],F,[X|Y1],Y2) :-
    X =< F,
    partition(Y,F,Y1,Y2).
partition([X|Y],F,Y1,[X|Y2]) :-
    X > F,
    partition(Y,F,Y1,Y2).

append([],Xs,Xs).
append([X|Xs],Ys,[X|Zs]) :-
    append(Xs,Ys,Zs).
```

Example: qsort

Ciao warns that it cannot verify that the call to =</2 will not generate a run-time error (assertion is in library!):

```
:- module(_,[qsort/2],[assertions,nativeprops,.(nmodes)]).

qsort([], []).
qsort([First|Rest],Result) :-
    partition(Rest,First,Sm,Lg),
    qsort(Sm,SmS),
    qsort(Lg,LgS),
    append(SmS,[First|LgS],Result).

partition([],_,[],[]).
partition([X|Y],F,[X|Y1],Y2) :-
 ➤ At literal 1 could not verify assertion:
    partition(Y,F,Y1,Y2).
partition([X|Y],F,Y1,[X|Y2]) :-
    X > F,
    partition(Y,F,Y1,Y2).

append([],Xs,Xs).
append([X|Xs],Ys,[X|Zs]) :-
    append(Xs,Ys,Zs).
```

Example: qsort

Adding useful entry information Ciao can infer that $=</2$ is called correctly, and no warnings are flagged (this would normally be obtained from analysis of caller to this module):

```
:- module(_,[qsort/2],[assertions,nativeprops,.(nmodes)]).

:- pred qsort(+list(num),_).

qsort([], []).
qsort([First|Rest],Result) :-
    partition(Rest,First,Sm,Lg),
    qsort(Sm,SmS),
    qsort(Lg,LgS),
    append(SmS,[First|LgS],Result).

partition([],_,[],[]).
partition([X|Y],F,[X|Y1],Y2) :-
    X =< F,
    partition(Y,F,Y1,Y2).
partition([X|Y],F,Y1,[X|Y2]) :-
    X > F,
    partition(Y,F,Y1,Y2).

append([],Xs,Xs).
append([X|Xs],Ys,[X|Zs]) :-
    append(Xs,Ys,Zs).
```

Example: qsort

We add some more assertions... :

```
:- pred qsort(+list(num),-list(num)) + is_det.

qsort([], []).
qsort([First|Rest],Result) :-
    partition(Rest,First,Sm,Lg),
    qsort(Sm,SmS),
    qsort(Lg,LgS),
    append(SmS,[First|LgS],Result).

:- pred partition(+list(num),+num,-list(num),-list(num)) + (is_det,not_fails).

partition([],_,[],[]).
partition([X|Y],F,[X|Y1],Y2) :-
    X =< F,
    partition(Y,F,Y1,Y2).
partition([X|Y],F,Y1,[X|Y2]) :-
    X > F,
    partition(Y,F,Y1,Y2).

:- pred append(+list(num),+list(num),-list(num)) + is_det.

append([],Xs,Xs).
append([X|Xs],Ys,[X|Zs]) :-
    append(Xs,Ys,Zs).
```

Example: qsort

...and they get verified by Ciao:

```
:- pred qsort(+list(num),-list(num)) + is_det.

qsort([], []).
qsort([First|Rest],Result) :-
    partition(Rest,First,Sm,Lg),
    qsort(Sm,SmS),
    qsort(Lg,LgS),
    append(SmS,[First|LgS],Result).

:- pred partition(+list(num),+num,-list(num),-list(num)) + (is_det,not_fails).

partition([],_,[],[]).
partition([X|Y],F,[X|Y1],Y2) :-
    X =< F,
    partition(Y,F,Y1,Y2).
partition([X|Y],F,Y1,[X|Y2]) :-
    X > F,
    partition(Y,F,Y1,Y2).

:- pred append(+list(num),+list(num),-list(num)) + is_det.

append([],Xs,Xs).
append([X|Xs],Ys,[X|Zs]) :-
    append(Xs,Ys,Zs).
```

Example: qsort

...and they get verified by Ciao:

```
:- pred qsort(+list(num),-list(num)) + is_det.

qsort([], []).
qsort([First|Rest],Result) :-
    partition(Rest,First,Sm,Lg),
    qsort(Sm,SmS),
    qsort(Lg,LgS),
    append(SmS,[First|LgS],Result).

:- pred partition(+list(num),+num,-list(num),-list(num)) + (is_det,not_fails).
➤ Verified assertion:
:- check comp partition(A,B,C,D)
   : ( list(num,A), num(B) )
   + ( is_det, not_fails ).
➤ Verified assertion:
:- check success partition(A,B,C,D)
   : ( list(num,A), num(B) )
   => ( list(num,C), list(num,D) ).

:- pred append(+list(num),+list(num),-list(num)) + is_det.

append([],Xs,Xs).
append([X|Xs],Ys,[X|Zs]) :-
    append(Xs,Ys,Zs).
```

Example: qsort

If we replace `=</2` with `</2` Ciao warns that `partition/3` can fail (cannot prove `not_fails`):

```
:- pred qsort(+list(num),-list(num)) + is_det.

qsort([], []).
qsort([First|Rest],Result) :-
    partition(Rest,First,Sm,Lg),
    qsort(Sm,SmS),
    qsort(Lg,LgS),
    append(SmS,[First|LgS],Result).

:- pred partition(+list(num),+num,-list(num),-list(num)) + (is_det,not_fails).

partition([],_,[],[]).
partition([X|Y],F,[X|Y1],Y2) :-
    X < F,
    partition(Y,F,Y1,Y2).
partition([X|Y],F,Y1,[X|Y2]) :-
    X > F,
    partition(Y,F,Y1,Y2).

:- pred append(+list(num),+list(num),-list(num)) + is_det.

append([],Xs,Xs).
append([X|Xs],Ys,[X|Zs]) :-
    append(Xs,Ys,Zs).
```

Example: qsort

If we replace `>=/2` with `>/2` Ciao warns that `partition/3` is not deterministic (cannot prove `is_det`):

```
:- pred qsort(+list(num),-list(num)) + is_det.

qsort([], []).
qsort([First|Rest],Result) :-
    partition(Rest,First,Sm,Lg),
    qsort(Sm,SmS),
    qsort(Lg,LgS),
    append(SmS,[First|LgS],Result).

:- pred partition(+list(num),+num,-list(num),-list(num)) + (is_det,not_fails).

partition([],_,[],[]).
partition([X|Y],F,[X|Y1],Y2) :-
    X =< F,
    partition(Y,F,Y1,Y2).
partition([X|Y],F,Y1,[X|Y2]) :-
    X >= F,
    partition(Y,F,Y1,Y2).

:- pred append(+list(num),+list(num),-list(num)) + is_det.

append([],Xs,Xs).
append([X|Xs],Ys,[X|Zs]) :-
    append(Xs,Ys,Zs).
```

Example: nrev (using the functional syntax package)

An example with more complex properties, a cost error is flagged:

```
:- module(_, [nrev/2], [assertions,fsyntax,nativeprops]).

:- pred nrev(A,B) : {list, ground} * var => list(B)
   + ( not_fails, is_det, steps_o( length(A) ) ).

nrev( [] )    := [].
nrev( [H|L] ) := ~conc( ~nrev(L),[H] ).


:- pred conc(A,B,C) + ( terminates, is_det, steps_o(length(A)) ).

conc( [],    L ) := L.
conc( [H|L], K ) := [ H | ~conc(L,K) ].
```

Example: nrev

Ciao reminds us that `nrev/2` is of course quadratic, not linear:

```
:- module(_, [nrev/2], [assertions,fsyntax,nativeprops]).

:- pred nrev(A,B) : {list, ground} * var => list(B)
 ➤ False assertion:
:- check comp nrev(A,B)
   : ( list(A), ground(A), var(B) )
   + ( not_fails, is_det, steps_o(length(A)) ).
because the comp field is incompatible with inferred comp:
[generic_comp] covered,is_det,mut_exclusive,not_fails,steps_lb(0.5*exp(length(A)
,2)+1.5*length(A)+1),steps_ub(0.5*exp(length(A),2)+1.5*length(A)+1)
 ➤ Verified assertion:
:- check calls nrev(A,B)
   : ( list(A), ground(A), var(B) ).
 ➤ Verified assertion:
:- check success nrev(A,B)
   : ( list(A), ground(A), var(B) )
   => list(B).
```

Example: nrev

With the cost expression fixed all properties are now verified:

```
:- module(_, [nrev/2], [assertions,fsyntax,nativeprops]).

:- pred nrev(A,B) : {list, ground} * var => list(B)
   + ( not_fails, is_det, steps_o( exp(length(A),2)) ) ).

nrev( [] )     := [].
nrev( [H|L] ) := ~conc( ~nrev(L),[H] ).


:- pred conc(A,B,C) + ( terminates, is_det, steps_o(length(A)) ).

conc( [],     L ) := L.
conc( [H|L], K ) := [ H | ~conc(L,K) ].
```

Example: nrev

If we change the assertion for `conc/3` from complexity order (`_o`) to upper bound (`_ub`) then Ciao flags that `length(A)` is not a correct upper bound:

```
:- module(_, [nrev/2], [assertions,fsyntax,nativeprops]).

:- pred nrev(A,B) : {list, ground} * var => list(B)
   + ( not_fails, is_det, steps_o( exp(length(A),2) ) ).

nrev( [] )     := [].
nrev( [H|L] ) := ~conc( ~nrev(L),[H] ).


:- pred conc(A,B,C) + ( terminates, is_det, steps_ub(length(A)) ).

conc( [],    L ) := L.
conc( [H|L], K ) := [ H | ~conc(L,K) ].
```

Example: nrev

If we change the assertion for `conc/3` from complexity order (_o) to upper bound (_ub) then Ciao flags that `length(A)` is not a correct upper bound:

```
:- module(_, [nrev/2], [assertions,fsyntax,nativeprops]).

:- pred nrev(A,B) : {list, ground} * var => list(B)
   + ( not_fails, is_det, steps_o( exp(length(A),2) ) ).

nrev( [] )     := [].
nrev( [H|L] ) := ~conc( ~nrev(L),[H] ).


:- pred conc(A,B,C) + ( terminates, is_det, steps_ub(length(A)) ).
➤ False assertion:
:- check comp conc(A,B,C)
   + ( terminates, is_det, steps_ub(length(A)) ).
because the comp field is incompatible with inferred comp:
[generic_comp] covered,is_det,mut_exclusive,not_fails,steps_lb(length(A)+1),step
s_ub(length(A)+1)
➤ Verified assertion:
:- check calls conc(A,B,C).
```

Example: nrev

With the cost expression fixed all properties are now verified:

```
:- module(_, [nrev/2], [assertions,fsyntax,nativeprops]).

:- pred nrev(A,B) : {list, ground} * var => list(B)
   + ( not_fails, is_det, steps_o( exp(length(A),2) ) ).

nrev( [] )     := [].
nrev( [H|L] ) := ~conc( ~nrev(L),[H] ).


:- pred conc(A,B,C) + ( terminates, is_det, steps_ub(length(A)+1) ).

conc( [],    L ) := L.
conc( [H|L], K ) := [ H | ~conc(L,K) ].
```

Example: nrev

With the cost expression fixed all properties are now verified:

```
:- module(_, [nrev/2], [assertions,fsyntax,nativeprops]).

:- pred nrev(A,B) : {list, ground} * var => list(B)
   + ( not_fails, is_det, steps_o( exp(length(A),2) ) ).

nrev( [] )     := [].
nrev( [H|L] ) := ~conc( ~nrev(L),[H] ).


:- pred conc(A,B,C) + ( terminates, is_det, steps_ub(length(A)+1) ).
➤ Verified assertion:
:- check calls conc(A,B,C).
➤ Verified assertion:
:- check comp conc(A,B,C)
   + ( terminates, is_det, steps_ub(length(A)+1) ).
```