GDC | March 20-24, 2023
San Francisco, CA

Logic Programming in Commercial Games: Experiences and Lessons Learned

Robert Zubek, SomaSim LLC, robert@somasim.com
Ian Horswill, Northwestern University, ian@northwestern.edu

Slides, software, and interactive demo:
https://tinyurl.com/game-lp

What we're going to talk about

- What is logic programming?
- LP in three commercial games
  - Asset consistency checking
  - Social graph queries
  - Procedural generation
- Experimental work

- Software, slides, discord, and interactive demo:
  https://tinyurl.com/game-lp

What is logic programming?

- ~~Logic~~
- ~~Declarative programming~~
- A **more declarative** language than C++/C#/etc.
  - Tell the system what you want
  - It decides how to do it
- Programming language based on **predicates**/**relations** and **rules**, rather than functions

Traditional languages

Basic unit is the **function**/method/procedure

- z = f(x,y)

Unidirectional

- Distinguish input(s) from output
- Always ask “here are all the inputs, what’s the output?”
- Reversing is hard

Work is done by **chaining calls**

Predicates (aka relations)

Generalization of functions

- F(x,y,z) means z=f(x,y)

Don’t prejudice what things are inputs or outputs

- You can set x and y and **solve for z**
- Or set y and z and **solve for x**
- **Set all of them** and just test if it’s true
- **Set none of them** and ask for a data point

Rules

A[x,y] if B[x,z], C[y,z]

“For any x,y,z, A[x,y] is true if B[x,z] and C[y,z] are true”

Most predicates are specified by rules

Example

```
Pet[x].If(Cat[x])
Pet[x].If(Dog[x])
Pet[x].If(Tiger[x], Tame[x])
```

"x is a pet if it's a cat, dog, or tame tiger"

Query: **`Pet[chris]`**

"is Chris a pet?"

Chris a pet if …

- They're a cat
- Or they're a dog
- Or they're a tiger …
  - … and also tame

Example

```
Pet[x].If(Cat[x])
Pet[x].If(Dog[x])
Pet[x].If(Tiger[x], Tame[x])
```

"x is a pet if it's a cat, dog, or tame tiger"

So in C#, this is like:

```
bool Pet(Mammal x)
   => Cat(x) || Dog(x)
      || (Tiger(x) && Tame(x))
```

Example

```
Pet[x].If(Cat[x])
Pet[x].If(Dog[x])
Pet[x].If(Tiger[x], Tame[x])
```

"x is a pet if it's a cat, dog, or tame tiger"

Query: **`Pet[x]`**

"find me a pet, x"

Set x to …

- A **cat**, if you can find one
- Otherwise, a **dog**
- Otherwise, a **tiger**
  - But check if it's **tame**
  - If not, try the **next tiger**

Example

```
Pet[x].If(Cat[x])
Pet[x].If(Dog[x])
Pet[x].If(Tiger[x], Tame[x])
```

"x is a pet if it's a cat, dog, or tame tiger"

So in C#, this is like:

```
Mammal? FindPet()
{
  Mammal? result = FindCat();

  if (result == null)
     result = FindDog();

  if (result == null)
     foreach (var x in AllTigers)
       if (IsTame(x))
       { result = x; break; }

  return result;
}
```

Example

```
Pet[x].If(Cat[x])
Pet[x].If(Dog[x])
Pet[x].If(Tiger[x], Tame[x])
```

"x is a pet if it's a cat, dog, or tame tiger"

Query: **`Pet[x].SolveForAll(x)`**

"find me all pets"

- List all the cats
- Then all the dogs
- Then, for each tiger
  - Check if it's tame
  - If so, it's a pet

Example

```
Pet[x].If(Cat[x])
Pet[x].If(Dog[x])
Pet[x].If(Tiger[x], Tame[x])
```

“x is a pet if it’s a cat, dog, or tame tiger”

So in C#, this is like:

```
IEnumerable<Mammal> FindPets()
  => FindCats().Cast<Mammal>()
     .Concat(FindDogs()).Cast<Mammal>()
     .Concat(FindTigers.Where(
              t => t.IsTame)
             .Cast<Mammal>);
```

Example

```
Pet[x].If(Cat[x])
Pet[x].If(Dog[x])
Pet[x].If(Tiger[x], Tame[x])
```

"x is a pet if it's a cat, dog, or tame tiger"

Query: **`Pet[x], Owner[x, rob]`**

"find me Rob's pet, x"

- Go through the pets, one by one (see Pet[x] previous)
- Check their owners
- Until you find Rob's pet (I'm guessing a tiger)

Example

```
Pet[x].If(Cat[x])
Pet[x].If(Dog[x])
Pet[x].If(Tiger[x], Tame[x])
```

"x is a pet if it's a cat, dog, or tame tiger"

Query: **`Owner[x, rob], Pet[x]`**

"find me Rob's pet, x"

- Go through Rob's stuff, item by item
- Check if each is a pet (see Pet[chris] example)

One rule
is worth many functions

Rules stand in for many different algorithms. The system chooses between them at run-time based on context

TELL: Typed, Embedded, Logic language

(github.com/ianhorswill/TELL)

```
Pet[x].If(Cat[x]);
Pet[x].If(Dog[x]);
Pet[x].If(Tiger[x], Tame[x]);
```

"x is a pet if it's a cat, dog, or tame tiger"

- Simple logic program subset
- Embedded in C#
  - TELL code **is C# code**
  - **Mix-and-match w/C#**
- Live coding support
- MIT license
- NB: Not highly optimized
- **Cheap and easy to experiment with**

TELL predicates are C# objects

```
var Pet = Predicate("Pet", x);
var Cat = Predicate("Cat", x);
var Dog = Predicate("Dog", x);
var Tiger = Predicate("Tiger", x);
var Tame = Predicate("Tame", x);

… rules for your fur babies …

Pet[x].If(Cat[x]);

Pet[x].If(Dog[x]);

Pet[x].If(Tiger[x], Tame[x]);

if (Pet[chris]) DoSomething();

var aPet = Pet[x].SolveFor(x);

var lotsOfPets = Pet[x].SolveForAll(x);
```

Methods for

- Adding rules
- Calling ([ ] is overloaded)
- Solving for variables

TELL variables are C# objects

```
Mammal chris = …;

var x = (Var<Mammal>)"x";
var Pet = Predicate("Pet", x);
var Cat = Predicate("Cat", x);
var Dog = Predicate("Dog", x);
var Tiger = Predicate("Tiger", x);
var Tame = Predicate("Tame", x);

… rules for your fur babies …

Pet[x].If(Cat[x]);
Pet[x].If(Dog[x]);
Pet[x].If(Tiger[x], Tame[x]);
if (Pet[chris]) DoSomething();
```

- **Strongly typed** (Var<int> vs. Var<Mammal>)
- Just represent a variable name, not its value
- **Predicates are also strongly typed**, based on variables passed in the constructor

Predicates can access game state

```
var Owner = Predicate<Person,Mammal>(
              "Owner",
              person -> person.Stuff);

var Tame = Predicate<Mammal>("Tame",
              ModeDispatch(
                // Argument is input
                m => m.IsTame,
                // Argument is output
                () => Mammal.AllMammals
                 .Where(m => m.IsTame)));
```

- Eventually, you want your rules to access your game state
- You do this with **primitive predicates**
- You just tell it "**run this C# code** when this predicate is called"
- Don't have time to go into it in detail, though

Reasons to use logic programming

- **Rules can be repurposed** for many different uses
- Easy to **slap a query language on your game state** (easier than SQL)
- Some **game logic is naturally described as rules** anyway

Flavors of logic programming

Top-down (Prolog)

- Start with a call to Pet, it tries each rule
- First rule calls Cat, second calls Dog, third calls Tiger and Tame

Bottom-up (DATALOG)

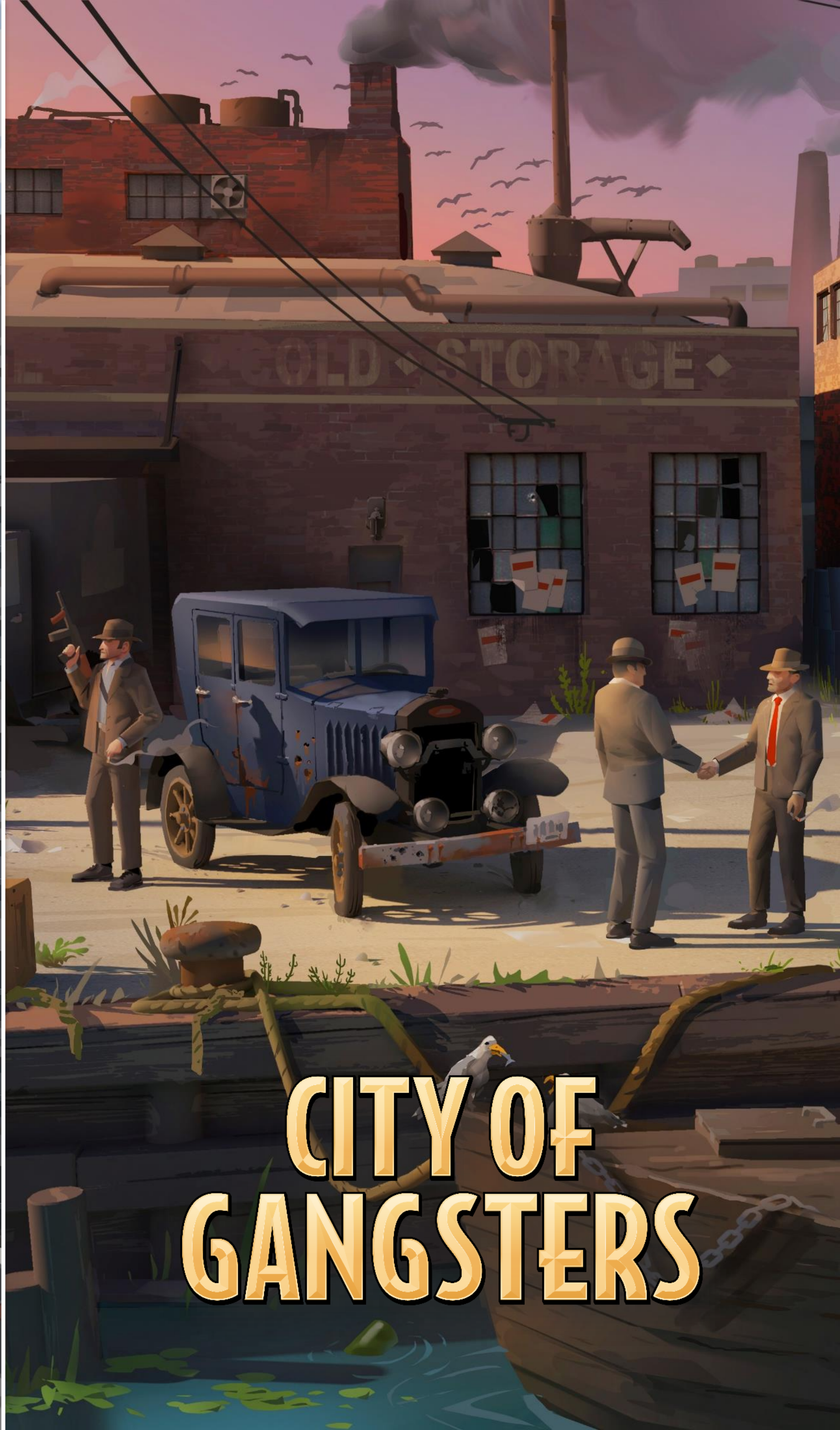
- Add all the cats to Pet, then all the dogs
- Then the intersection of Tiger and Tame

SAT-based (ASP, SMT)

- Use general constraint satisfaction algorithms


1849


SomaSim
PROJECT
HIGHRISE


CITY OF
GANGSTERS

LP systems

PROJECT HIGHRISE ⇨ Unity Prolog

CITY OF GANGSTERS ⇨ BotL, CatSAT

(Future game) ⇨ TELL & TED, CatSAT

Project Highrise

Skyscraper simulator

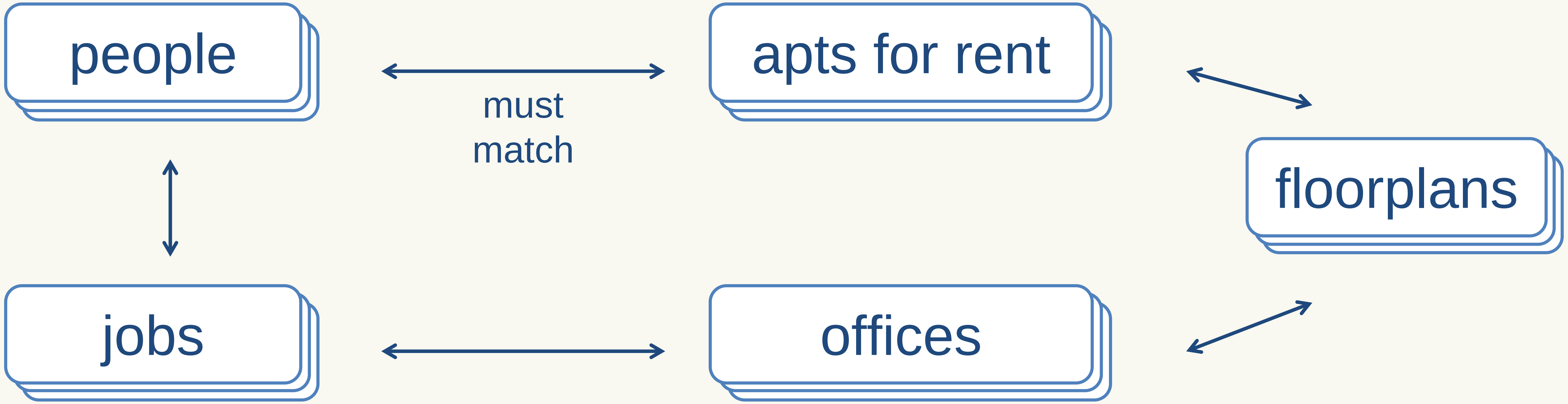
Two AI experiments:

- AI Planner for NPC behavior → see [Game AI Pro 3 article]
- Prolog for asset consistency checking → **Unity Prolog!**

Consistency Checking

Example: we have tons of assets with cross-references
xrefs use **names** and **tags** for flexibility

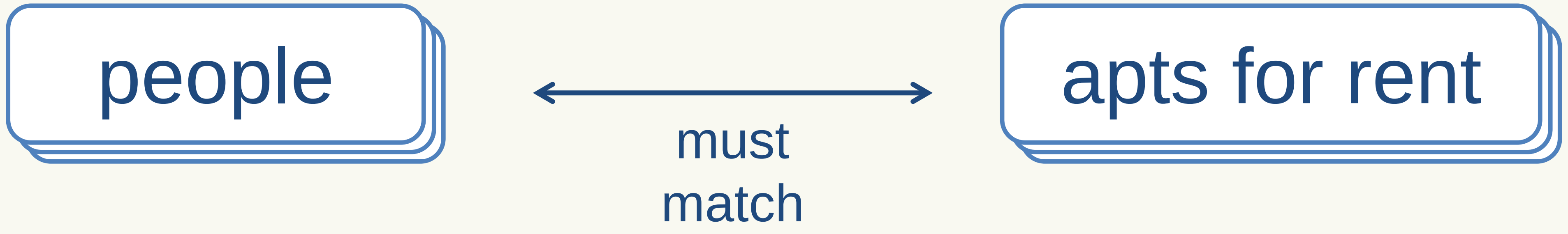
Problem: make sure all things are matched to each other



Consistency Checking

Matching residents with apartments for rent

- Space tags (what they are): `[ apartment, 1br, berlin ]`
- People tags (what they want): `[ 1br, 2br, … ]`



Consistency Checking

Verify that all tags are set up correctly:

- all spaces have someone who could live there
- all people have a space they want to live in

You *could* do this with tons of **foreach** loops…

```
MatchTags(S,P):
  for each tag T in tagsof(S)
    for each tag T' in tagsof(P)
      return true if T = T'

for each space S
  for each people P
    signal a problem if MatchTags(S,P) != true
```

Consistency Checking

Verify that all tags are set up correctly:

- all spaces have someone who could live there
- all people have a space they want to live in

You *could* do this with tons of **foreach** loops, or…
express it as a **rule** and
let the computer validate it

Rule:

$\forall\ S \in \text{Spaces},\ P \in \text{People}$

$\exists\ T{:}\ T \in \text{tagsof}(S) \land T \in \text{tagsof}(P)$

Signal a problem if false

Prolog code

```
problems :-
    step_limit(10000000),
    all(P, problem(P), Problems),
    forall(member(P, Problems), writeln(P)).

problem(P) :-
    unit(U),
    unit_problem(U, P).

problem(P) :-
    workplace(W),
    workplace_problem(W, P).

problem(P) :-
    residence(R),
    residence_problem(R, P).
```

```
residence_problem(R, no_matching_movein(R)) :-
    \+ matches_residential_movein(R, _).

matches_residential_movein(R, M) :-
    RTags is R.unit.tags,
    MoveIns is $'Game'.serv.globals.settings.economy.moveins,
    member(M, MoveIns),
    member(Tag, RTags),
    element(Tag, M.instant.tags).
```

Iterates through all residence definitions,
all peoples' move-in preferences,
all their respective tags,
and makes sure they match up

Prolog code

```
problems :-
    step_limit(10000000),
    all(P, problem(P), Problems),
    forall(member(P, Problems), writeln(P)).

problem(P) :-
    unit(U),
    unit_problem(U, P).

problem(P) :-
    workplace(W),
    workplace_problem(W, P).

problem(P) :-
    residence(R),
    residence_problem(R, P).
```

We ended up with
400 lines of Prolog

Most of it written
in one afternoon

Takeaways

Tests were **queries over tree-like data structures**

- Loved writing tests as queries, rather than imperatively
- All tests could be localized in a central place

Unity Prolog

- ISO Prolog – very powerful, but can be tricky
- Need to understand how queries get executed (e.g. “cut”)
- Our use cases didn’t exercise all that power

City of Gangsters

Prohibition-era
organized crime sim

Full of secret deals
and vendettas

AI task: querying over social graph at runtime

City of Gangsters

We wanted social effects like vendettas

"You killed my father, prepare to die"

Run a query over social graph:

- Find **X, Y, Z** such that
- **X** is the player
- **X killed Y**
- **Y** is **Z**'s **relative**

And modify **Z**'s relationship to **X**

City of Gangsters

Also, nice effects:

"You helped my friend, I appreciate that"

Run a query over social graph:

- Find **X, Y, Z** such that
- **X** is the player
- **X helped Y**
- **Y** is **Z**'s **friend**

And modify **Z**'s relationship to **X**

City of Gangsters

Or more generally, social norms:

"You *[acted]* on *[someone's] [relation]*, I have *[reaction]*"

Run a query over social graph:

- Find **X, Y, Z, A**, **R** such that
- **X** is the player
- **X** ***performed action A*** *with* **Y**
- **Y** and **Z** ***have relationship R***

And modify **Z**'s relationship to **X**

Positive examples: help, complete quest, give money

Negative: extort, harm, kill

BotL, aka "Bot Language"

C# implementation, highly optimized for runtime performance

- Stack allocated, *no runtime mallocs*
- Custom VM: *Vienna Abstract Machine 2P*
- Prolog feature set cut down to help with performance

BotL code

```
socialInference(OtherPeep, TargetPeep, HumanPeep, Link, "violence", "violence-inf") <--
    hasFamilyHistory(OtherPeep, TargetPeep, HumanPeep, Link, "violence");


// hasAnyFriendHistory(+OtherPeep, +TargetPeep, +HumanPeep, -Link, +Action)
hasAnyFriendHistory(OtherPeep, TargetPeep, HumanPeep, AnyLink, Action) <--
    findLink(OtherPeep, TargetPeep, AnyLink),
    findHistory(TargetPeep, HumanPeep, History),
    findActionInHistory(History, Action);

// findLink(+Source, +Target, ?SocLink)
findLink(Source, Target, SocLink) <--
    findHistory(Source, Target, History),
    SocLink = History.link;

// findActionInHistory(+History, +Action)
findActionInHistory(History, Action) <--
    History.ContainsAction(Action) = true;
```

Actual BotL code is not super readable :)

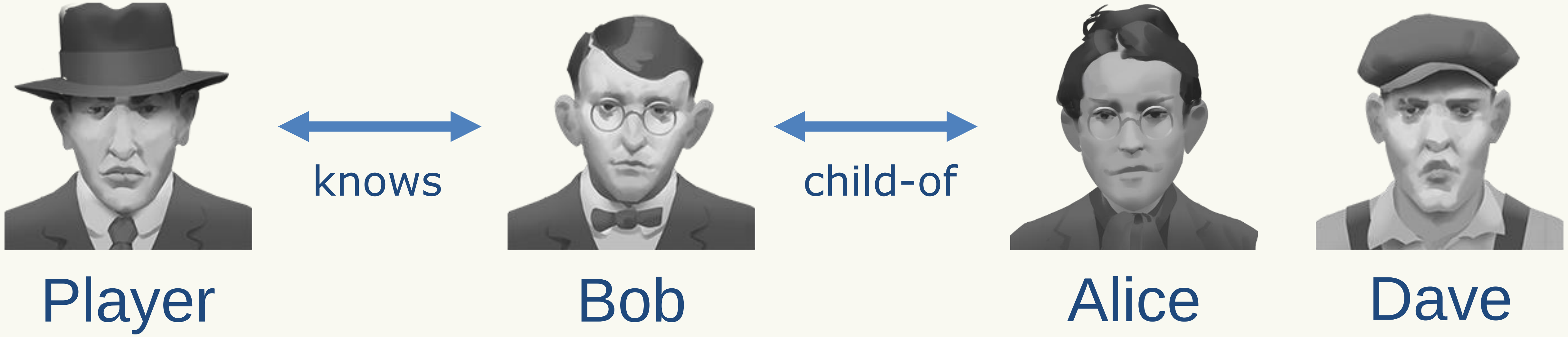
Let's rephrase a bit…

Social inference in TELL

(MicroCoG demo at https://tinyurl.com/game-lp)

```
var ReactionToAction =
    Predicate(action, target, reactor, reaction, buffTotal)

      .If(ReactionType[action, reaction, relationshipClass],
          RelationshipOf[target, rel],
          RelationshipClass[rel, relationshipClass],
          RelationshipType[rel, relationshipType],
          RelationshipTo[rel, reactor],
          Sum[buff,
              ReactionBuff[reaction, relationshipClass,
                           relationshipType, buff],
              buffTotal]);
```

Social inference




```
action: violence
target: Bob
```

Social inference

If somebody does **action** to NPC **target**

```
var ReactionToAction =
    Predicate(action, target, reactor, reaction, buffTotal)

      .If(ReactionType[action, reaction, relationshipClass],
          RelationshipOf[target, rel],
          RelationshipClass[rel, relationshipClass],
          RelationshipType[rel, relationshipType],
          RelationshipTo[rel, reactor],
          Sum[buff,
              ReactionBuff[reaction, relationshipClass,
                           relationshipType, buff],
              buffTotal]);
```

Social inference

Then NPC **reactor** will have **reaction**

```
var ReactionToAction =
    Predicate(action, target, reactor, reaction, buffTotal)

        .If(ReactionType[action, reaction, relationshipClass],
            RelationshipOf[target, rel],
            RelationshipClass[rel, relationshipClass],
            RelationshipType[rel, relationshipType],
            RelationshipTo[rel, reactor],
            Sum[buff,
                ReactionBuff[reaction, relationshipClass,
                             relationshipType, buff],
                buffTotal]);
```

Social inference

And it will change their trust by **buffTotal**

```
var ReactionToAction =
    Predicate(action, target, reactor, reaction, buffTotal)

      .If(ReactionType[action, reaction, relationshipClass],
          RelationshipOf[target, rel],
          RelationshipClass[rel, relationshipClass],
          RelationshipType[rel, relationshipType],
          RelationshipTo[rel, reactor],
          Sum[buff,
              ReactionBuff[reaction, relationshipClass,
                           relationshipType, buff],
              buffTotal]);
```

Social inference

IF

```
var ReactionToAction =
    Predicate(action, target, reactor, reaction, buffTotal)

      .If(ReactionType[action, reaction, relationshipClass],
          RelationshipOf[target, rel],
          RelationshipClass[rel, relationshipClass],
          RelationshipType[rel, relationshipType],
          RelationshipTo[rel, reactor],
          Sum[buff,
              ReactionBuff[reaction, relationshipClass,
                           relationshipType, buff],
              buffTotal]);
```

Social inference

Reaction is the kind of reaction people have to that **action**

```
var ReactionToAction =
    Predicate(action, target, reactor, reaction, buffTotal)

      .If(ReactionType[action, reaction, relationshipClass],
          RelationshipOf[target, rel],
          RelationshipClass[rel, relationshipClass],
          RelationshipType[rel, relationshipType],
          RelationshipTo[rel, reactor],
          Sum[buff,
              ReactionBuff[reaction, relationshipClass,
                           relationshipType, buff],
              buffTotal]);
```

Social inference

Rel is someone in in **target**'s network of the right **class**

```
var ReactionToAction =
    Predicate(action, target, reactor, reaction, buffTotal)

      .If(ReactionType[action, reaction, relationshipClass],
          RelationshipOf[target, rel],
          RelationshipClass[rel, relationshipClass],
          RelationshipType[rel, relationshipType],
          RelationshipTo[rel, reactor],
          Sum[buff,
              ReactionBuff[reaction, relationshipClass,
                           relationshipType, buff],
              buffTotal]);
```

Social inference

Rel is of this **type** (mother, acquaintance, …)

```
var ReactionToAction =
    Predicate(action, target, reactor, reaction, buffTotal)

      .If(ReactionType[action, reaction, relationshipClass],
          RelationshipOf[target, rel],
          RelationshipClass[rel, relationshipClass],
          RelationshipType[rel, relationshipType],
          RelationshipTo[rel, reactor],
          Sum[buff,
              ReactionBuff[reaction, relationshipClass,
                                     relationshipType, buff],
              buffTotal]);
```

Social inference

Rel is a relationship of the **target** to the **reactor**

```
var ReactionToAction =
    Predicate(action, target, reactor, reaction, buffTotal)

      .If(ReactionType[action, reaction, relationshipClass],
          RelationshipOf[target, rel],
          RelationshipClass[rel, relationshipClass],
          RelationshipType[rel, relationshipType],
          RelationshipTo[rel, reactor],
          Sum[buff,
              ReactionBuff[reaction, relationshipClass,
                           relationshipType, buff],
              buffTotal]);
```

Social inference

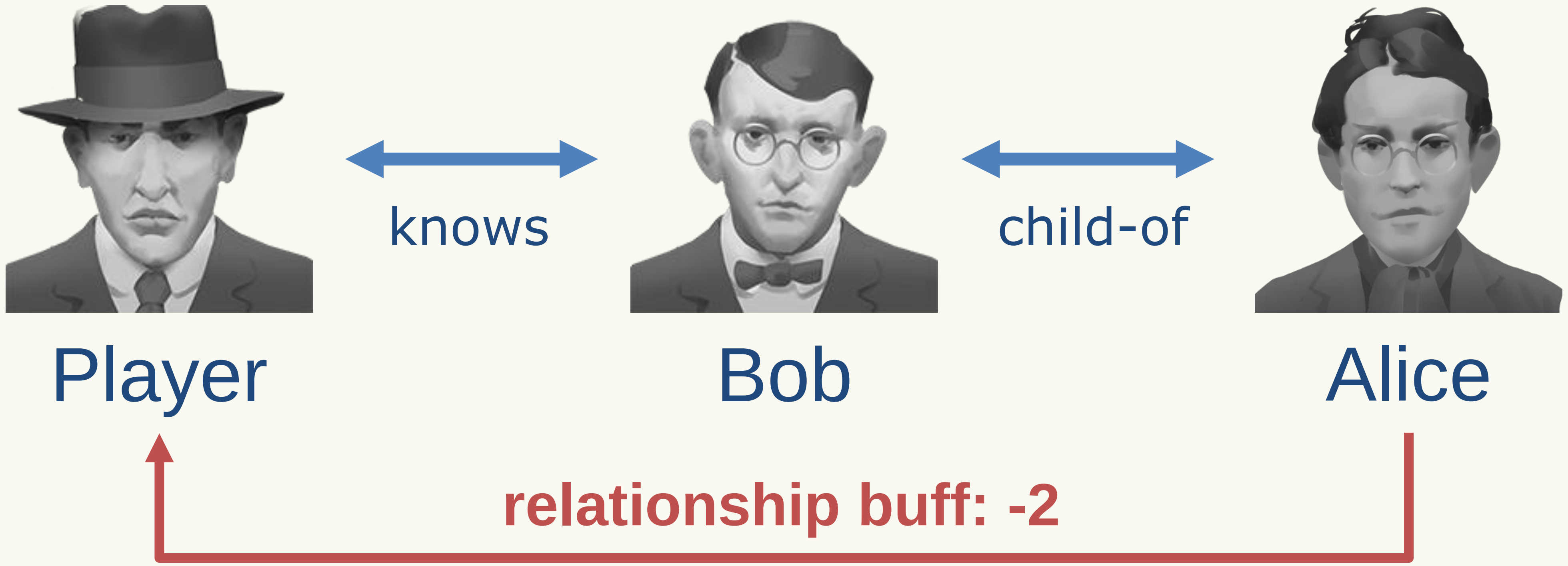
And **buffTotal** is the sum of all the applicable **buffs**

```
var ReactionToAction =
    Predicate(action, target, reactor, reaction, buffTotal)

      .If(ReactionType[action, reaction, relationshipClass],
          RelationshipOf[target, rel],
          RelationshipClass[rel, relationshipClass],
          RelationshipType[rel, relationshipType],
          RelationshipTo[rel, reactor],
          Sum[buff,
              ReactionBuff[reaction, relationshipClass,
                           relationshipType, buff],
              buffTotal]);
```

Social inference

```
var ReactionToAction =
    Predicate(action, target, reactor, reaction, buffTotal)

      .If(ReactionType[action, reaction, relationshipClass],
          RelationshipOf[target, rel],
          RelationshipClass[rel, relationshipClass],
          RelationshipType[rel, relationshipType],
          RelationshipTo[rel, reactor],
          Sum[buff,
              ReactionBuff[reaction, relationshipClass,
                           relationshipType, buff],
              buffTotal]);
```

Social inference in TELL




```
action: violence
target: Bob
reactor: Alice
reaction: violence-reaction
relationshipClass: Family
relationshipType: Child
buff: -2
```

Takeaways

Declarative queries are great

BotL is really, really fast

We wished for better debugging and embedding into C#

Idea: but what if we had LP that's more like Linq than SQL?

- Embedded in C# rather than external
- Use strong typing and easy .NET interop
- Benefit from Visual Studio debugger, IntelliSense, etc.

(Future game prototype)

Consistency checking

Similar problems as in Project Highrise

Using two new systems: TELL and TED

- TELL: Typed Embedded Logic Language
- TED: Typed Embedded Datalog

Consistency checking

Example: there are definitions for **companies** and **contracts,** matched by tags. Make sure everybody matches up.

$$\begin{aligned}&\text{invalidCompany}(C, Id) \Leftarrow\\&\quad \text{isCompany}(C) \land\\&\quad \text{offersContract}(C, Id) \land\\&\quad \neg\, \text{isContract}(Id)\end{aligned}$$

$$\begin{aligned}&\text{orphanedContract}(Id) \Leftarrow\\&\quad \text{isContract}(Id) \land\\&\quad \nexists C: \text{offersContract}(C, Id)\end{aligned}$$

TED code

```
var is_company = Predicate(listOfCompanies);
var is_contract = Predicate(listOfContracts.Select(a => a.id));

var company_offers_contract = RelationFromMemberList(...);
var contract_offered_by_anyone = Predicate(...);

var bad_company_contract = Predicate(Company, Contract).If(
    is_company[Company],
    company_offers_contract[Company, Contract],
    !is_contract[Contract]
    );

Log2("This company is offering an invalid contract", bad_company_contract);

var orphaned_contract = Predicate(Contract).If(
    is_contract[Contract],
    !contract_offered_by_anyone[Contract]);

Log("This contract is not offered by any company", orphaned_contract);
```

invalidCompany(C, Id) ⇐
isCompany(C) ∧
offersContract(C, Id) ∧
¬ isContract(Id)

orphanedContract(Id) ⇐
isContract(Id) ∧
∄C: offersContract(C, Id)

TED code

```
var is_company = Predicate(listOfCompanies);
var is_contract = Predicate(listOfContracts.Select(a => a.id));

var company_offers_contract = RelationFromMemberList(...);
var contract_offered_by_anyone = Predicate(...);

var bad_company_contract = Predicate(Company, Contract).If(
    is_company[Company],
    company_offers_contract[Company, Contract],
    !is_contract[Contract]
    );

Log2("This company is offering an invalid contract", bad_company_contract);

var orphaned_contract = Predicate(Contract).If(
    is_contract[Contract],
    !contract_offered_by_anyone[Contract]);

Log("This contract is not offered by any company", orphaned_contract);
```

Each predicate can *test* a specific value, or *generate* all matching values

Here is_contract[] tests a specific value of variable Contract

Here is_contract[] produces all possible values for Contract

TELL vs TED

Different execution strategies!

TELL

- Like Prolog, execute queries top-down, depth-first search

TED

- Like Datalog, execute queries bottom-up, creating a table for each predicate or expression, and merging them

Takeaways

Very early in development, but:

- Embedding inside C# is very, very, very nice
- No longer purely stack-allocated, but that's okay
- Both systems in active development:
  - Optimizations coming
  - TED should be easy to parallelize

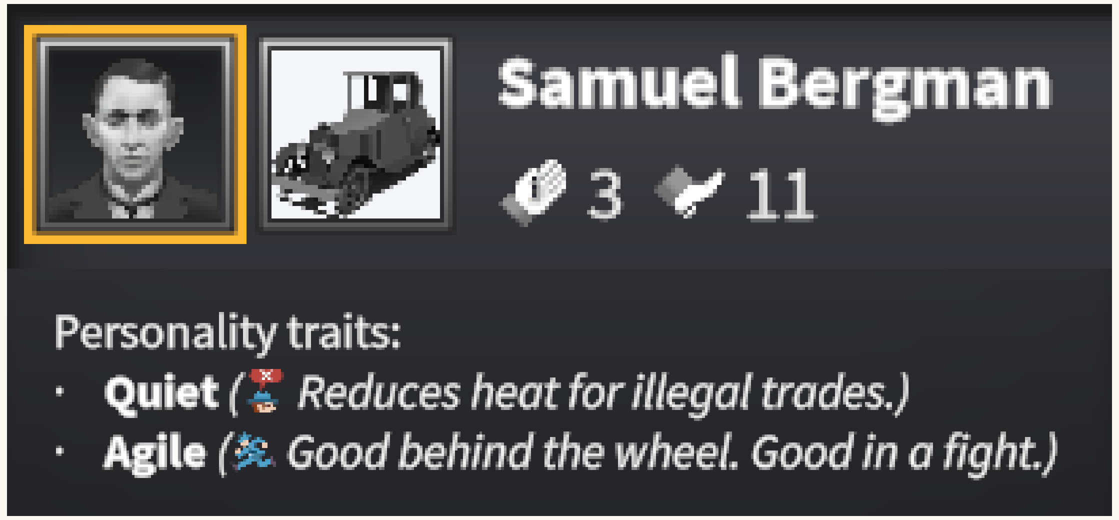
LP systems we used

System	Embedded language	Search	Type	Memory allocation
Unity Prolog	No	Top-down	ISO Prolog	Dynamic
BotL	No	Top-down	Prolog-like	Stack-based
TELL	Yes	Top-down	Prolog-like	Dynamic
TED	Yes	Bottom-up	Datalog-like Parallelizable	Dynamic

One more thing… NPC procgen!

Two use cases

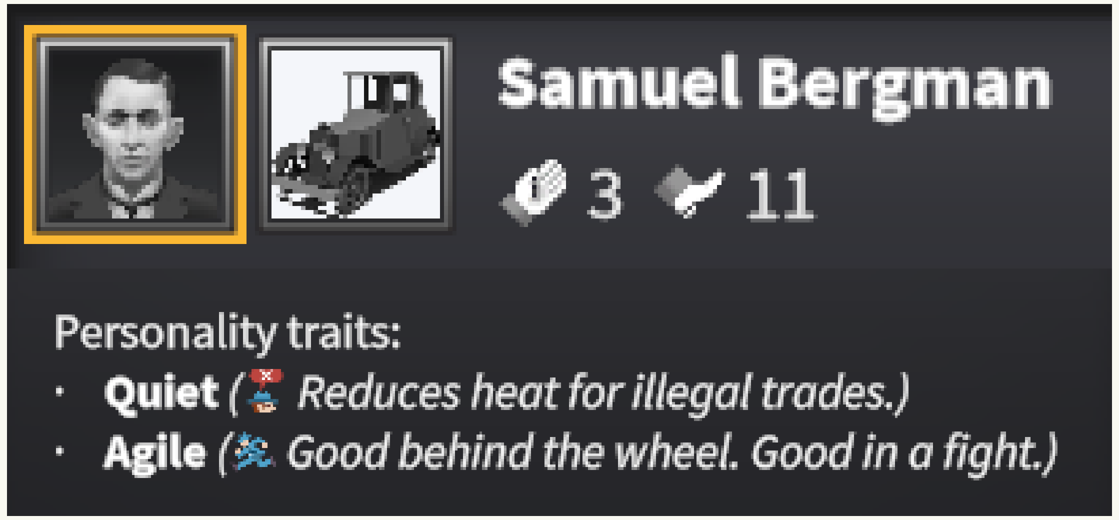
1. NPC personality traits
   (City of Gangsters, future game)



2. NPC composite portraits
   (future game)

Two use cases

1. NPC personality traits



Pick 3 traits that together satisfy design constraints:

$$loner \rightarrow \neg\, sociable \wedge quiet$$
$$quiet \rightarrow \neg\, talkative$$
$$talkative \rightarrow \neg\, quiet \wedge friendly$$
$$agile \rightarrow \neg\, clumsy \wedge \neg\, lazy$$

$$\ldots etc.$$

Two use cases

1. NPC personality traits
2. NPC composite portraits

Pick 1 body, head, hair, etc. that satisfy art constraints:

$$male \leftrightarrow head_1 \vee \cdots \vee head_M$$
$$male \leftrightarrow body_1 \vee \cdots \vee body_N$$
$$body_5 \leftrightarrow head_4 \vee head_6$$
$$head_3 \vee head_4 \rightarrow hair_8 \vee hair_9 \vee hair_{10}$$
$$hair_8 \rightarrow \neg\, head_{10}$$
$$\ldots etc.$$

Approach

It’s a **satisfiability** problem!

- Find a **model** (set of true values) that **satisfies** all those constraints
- SAT solvers exist…

Approach

It's a **satisfiability** problem!

- Find a **model** (set of true values) that **satisfies** all those constraints
- SAT solvers exist…

CatSAT

SAT solver with **randomization**

- Better randomization of solutions than traditional SAT solvers
- Randomization is key for PCG!

Implemented in C#

Experimental systems

PCG for non-programmers, creative writers, and tabletop GMs

Imaginarium

Constraint-based PCG from English-language descriptions

Persian, tabby, Siamese, manx, Chartreux, and Maine coon are kinds of cat.

Cats are long-haired or short-haired.

Cats can be big or small.

Chartreux are grey.

Siamese are grey.

Persians are long-haired.

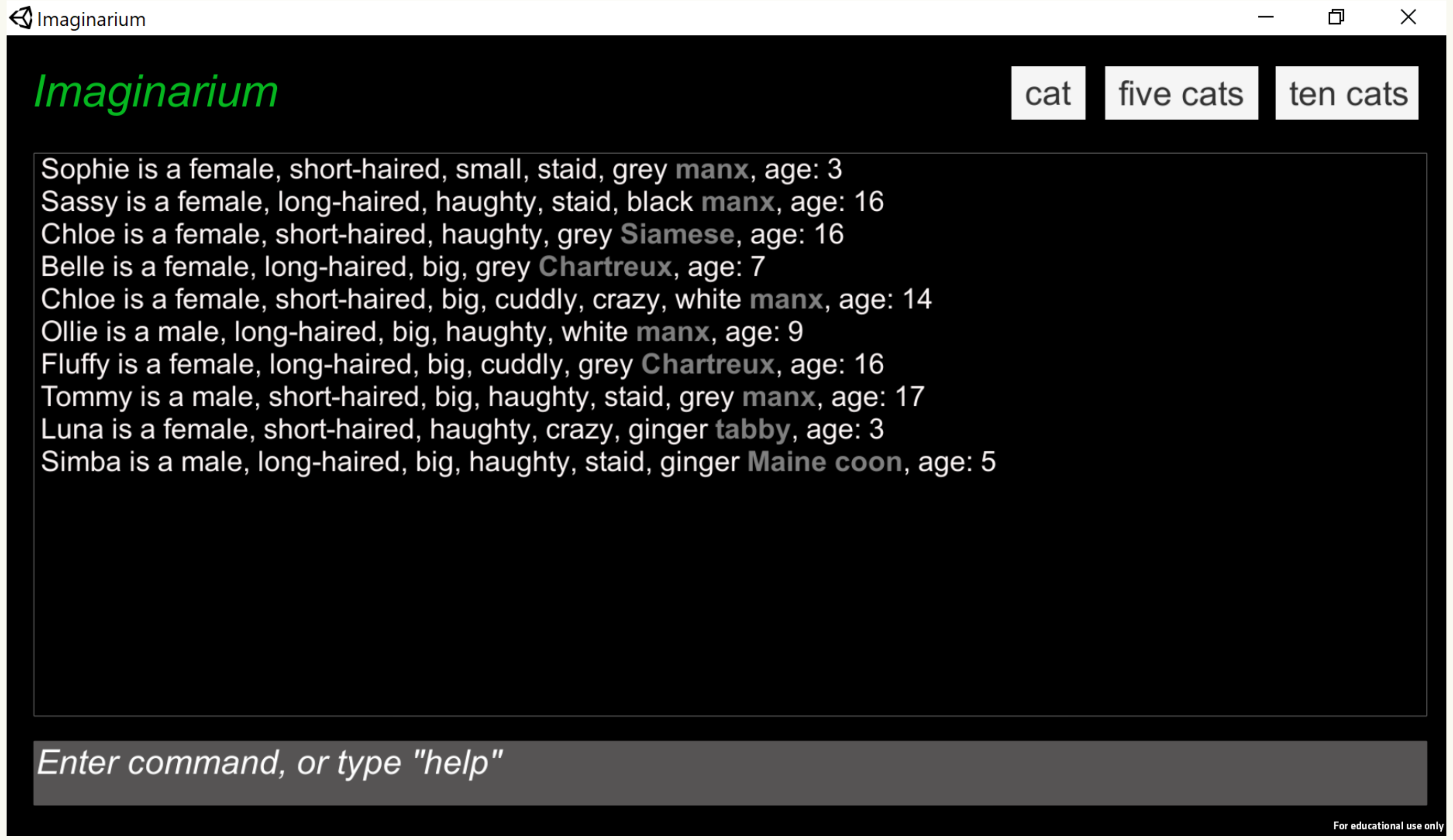
Siamese are short-haired.

Maine coons are large.

Cats are black, white, grey, or ginger.

…

imagine 10 cats



Imaginarium

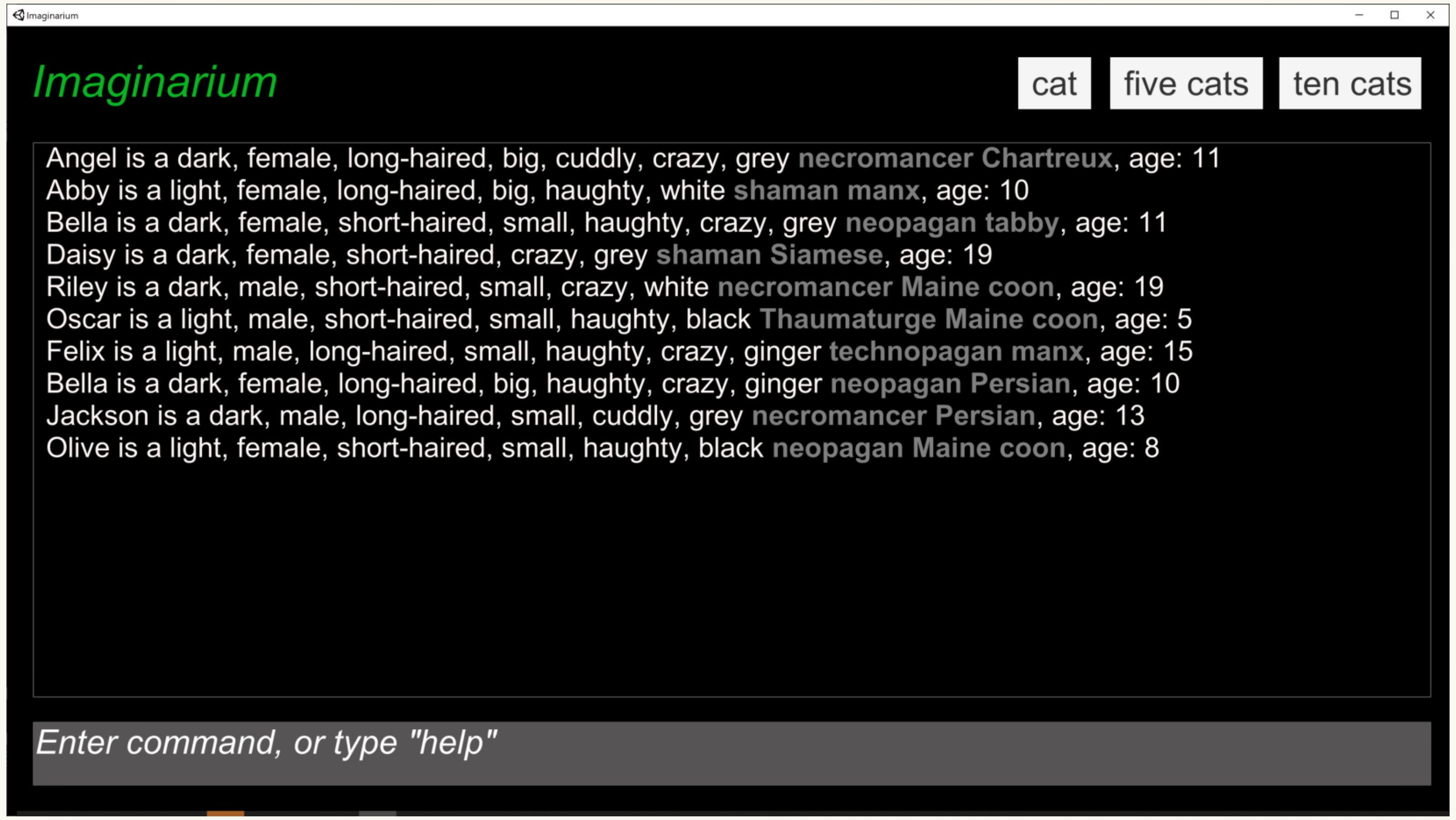
Constraint-based PCG from English-language descriptions

Thaumaturge, **necromancer**, **neopagan**, **technopagan**, and **shaman** are kinds of **magic user**.

A magic user is **dark or light**

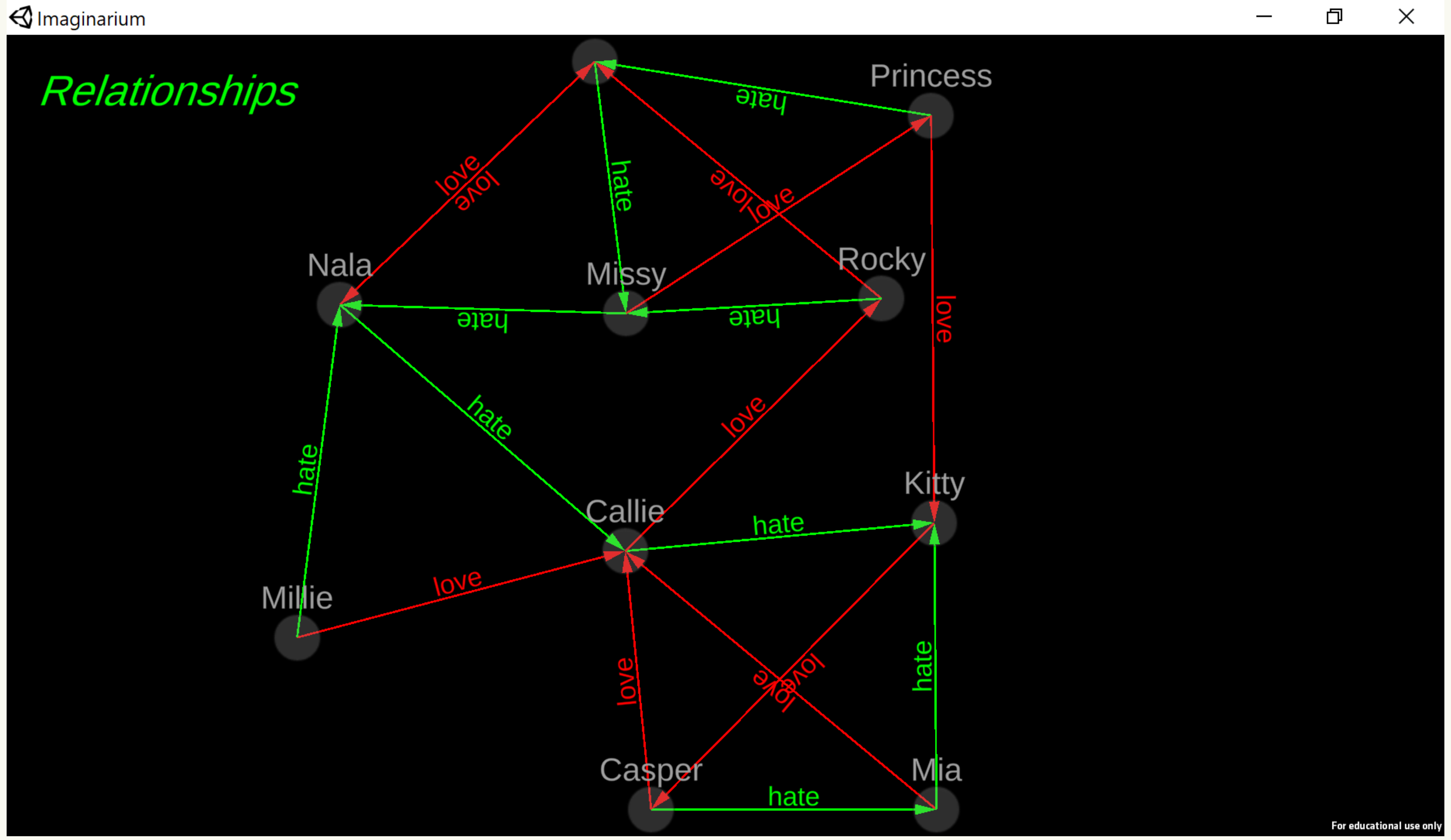
Necromancers are dark
Thaumaturges are light

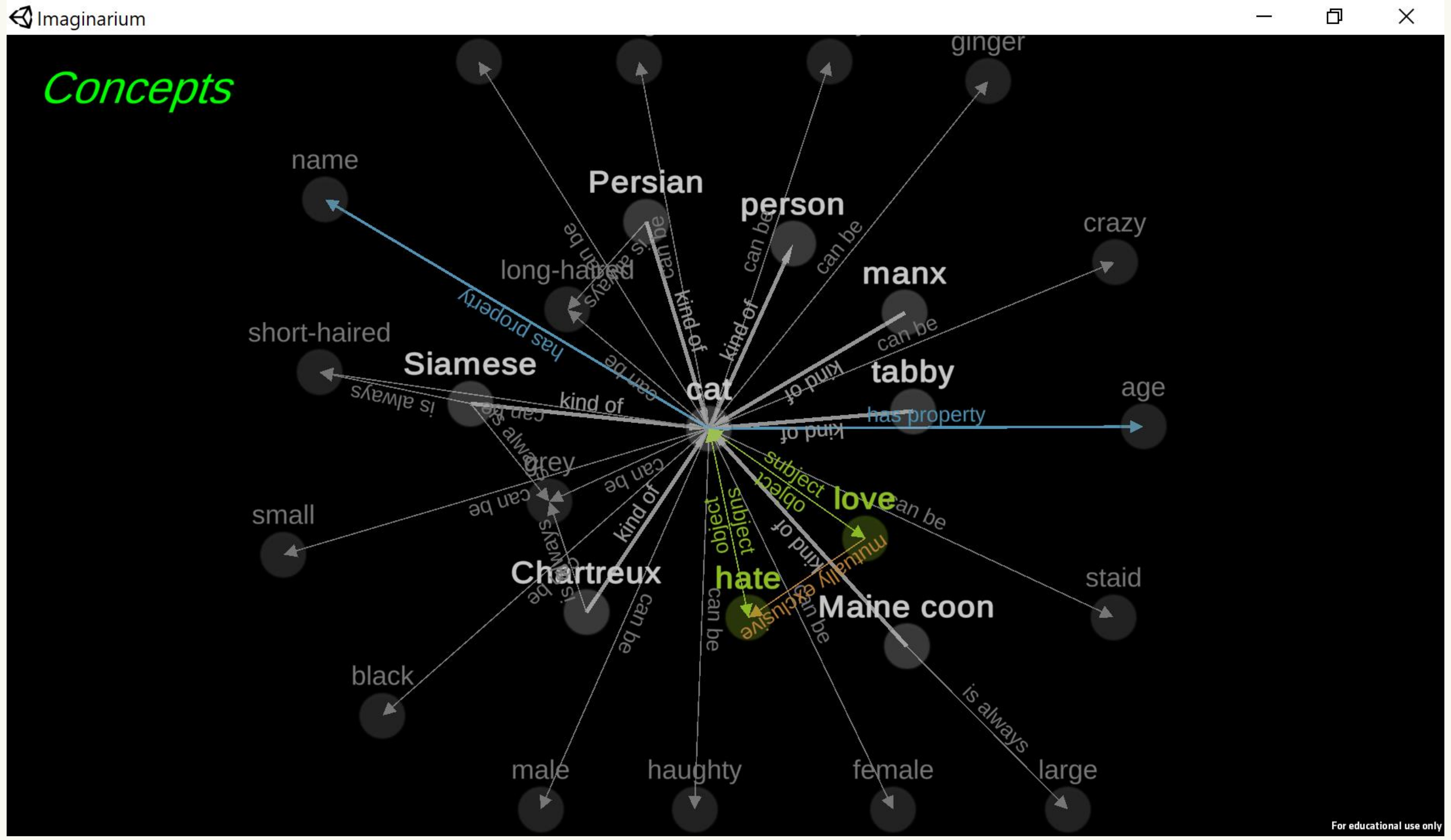
imagine 10 magic user cats



Imaginarium

Generating and visualizing relationships





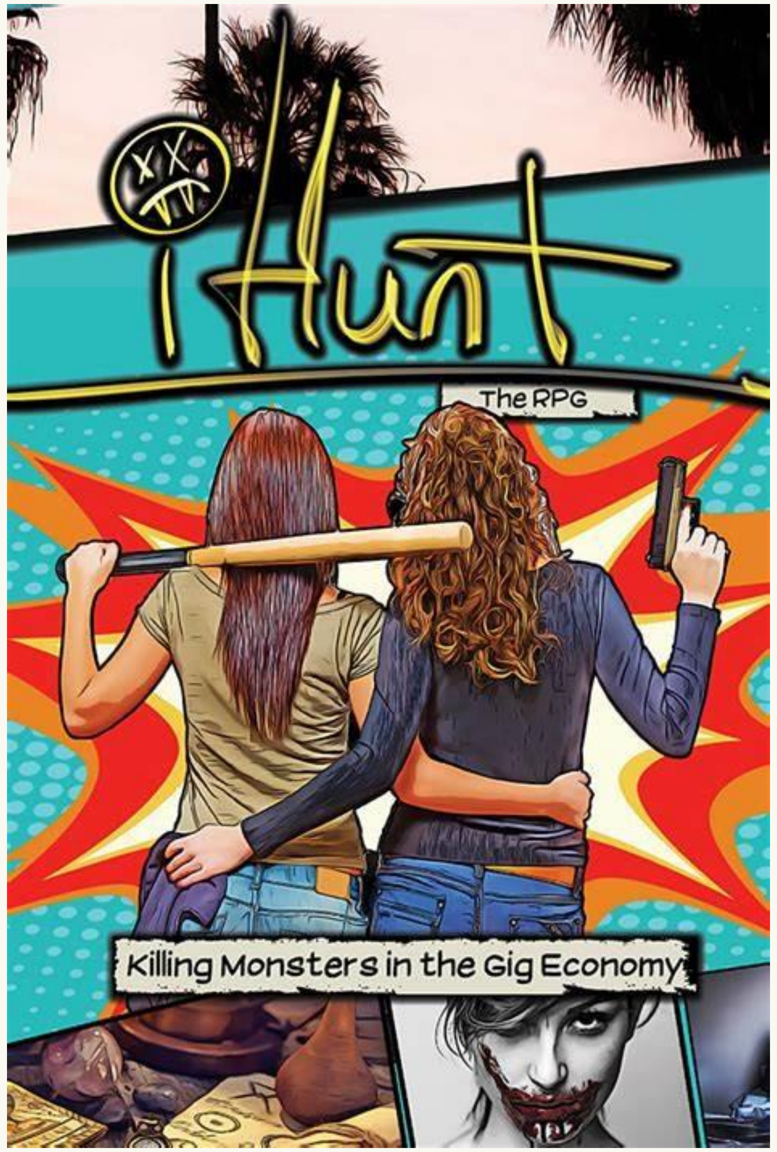
Generative text for TTRPGs

(Joint work with Olivia Hill and Filamena Young)

You are hired by a curious party to take out a werewolf. The client just really hates werewolves.

When you finally find the werewolf, the area is swarming with cops. And they turn out to be extremely attractive. What will you do?

Afterward, the client has another job for you, a really hard one, and they want you to start right now.



Summary

Logic programming is great for specific purposes

- Focuses on query, hides execution strategy

Prolog/Datalog-likes: used as “SQL for knowledge graphs”

Satisfiability solver: used for PCG with constraints

Ergonomic implementations are key!

Links

All these systems are open source! (MIT license)

Go here: **https://tinyurl.com/game-lp**

- Newer systems: TED, TELL, CatSAT
- Older systems: BotL, Unity Prolog
- Sample app

Also, join us on Discord! Talk about LP in games, get code feedback, and more. Link at: https://tinyurl.com/game-lp

Thank you!

Robert Zubek, SomaSim LLC, robert@somasim.com
Ian Horswill, Northwestern University, ian@northwestern.edu

Slides, software, and interactive demo:
https://tinyurl.com/game-lp