Andrew Blinn

Kevin Li

Cyrus Omar

FPLab @ UMich

Type-directed Prompt Construction for LLM-powered Programming Assistants

October 2023 Experience Report

Hazel Demo

hazel Examples Void

```
#Render main TODO view#
#Should be a brown box with rounded corners and nice padding #
#Labelled Hazel TODOs. Use comics sans#
let view: (Action -> Event, Model) -> Node =
  fun go, (buffer, todos) ->
    Div(
      [
        Style([
          BackgroundColor("#946"),
          BorderRadius("0.3em"),
          Padding("1em")]),
        Create("class", "app")
      ],
      [
        Div([
          Style([
            FontFamily("'Comic Sans MS', cursive, sans-serif"),
            Color("#fff"),
            Margin("1em"),
            Display("flex"),
            JustifyContent("center")]),
          Create("class", "header")
        ], [Text("Hazel TODOs")]),
        buffer(go),
        add_button(go),
        todos_deck(go,todos)
      ]
    ) in
```

Γ EXP ? Function literal : (Action -> Event, Model) -> Node equals expected type

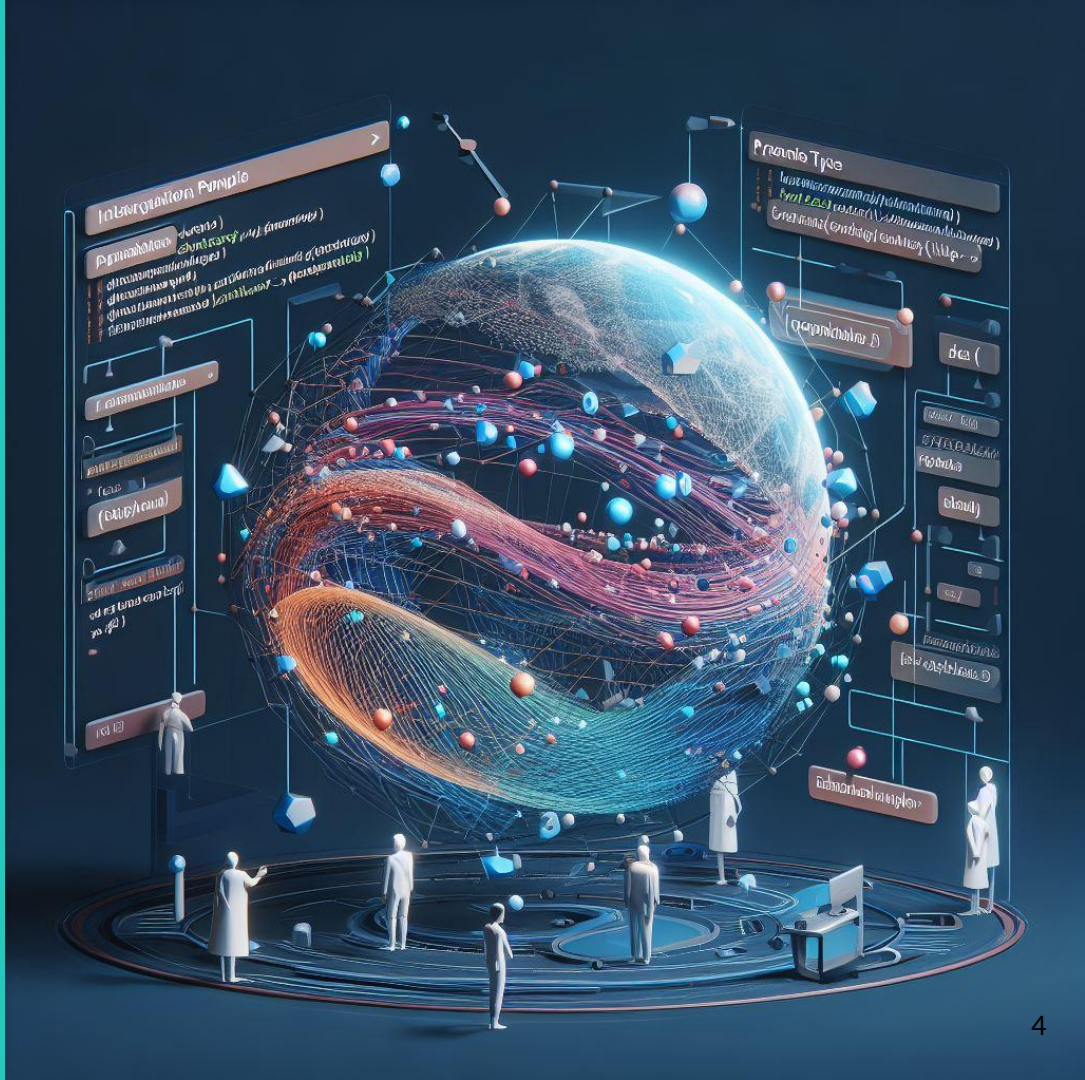
Intro

- Large language models have changed the landscape for code completion

- But how do they get fed? Poor or missing information can lead to hallucination

- We have a wealth of semantic information available that can be used to inform and constrain generation

LLMs vs Types



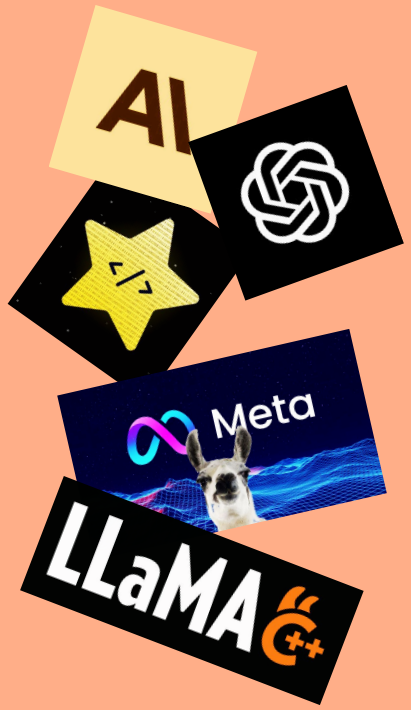
Types & LLMs: Opportunities for Intervention

- **Before prompting**
  Types informing prompt creation
- **After prompting**
  Types for validation + correction

Types & LLMs: Opportunities for Intervention

- **Before prompting**
  Types informing prompt creation
- **After prompting**
  Types for validation + correction
- **During prompting** **(Future work)**
  Types constraining productions by-token
- **Between prompting** **(Future work)**
  Types informing multi-step strategies

Hazel LLM Assistant



LLMs for code completion: The model landscape

- We're PL people, not ML people. As such we're mostly using existing models as black boxes

Commercial APIs

- Smart (out of the box)
- Convenient

Local models

- Flexible
- Reproducible

LLMs for code completion: The model landscape

- For our initial experiments, we picked OpenAI GPT4 via Microsoft Azure

- We were able to quickly generate mostly syntactically correct Hazel code via basic prompt engineering; not true for GPT3.5 or llama

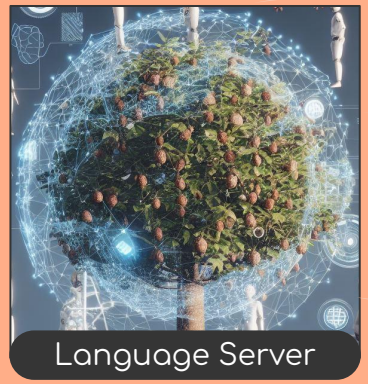
Without type info

- Certain examples are awful
- Ocaml vscode copilot
- Hallucination without type info
- Try in vscode

Hazel Intro

- Maybe not necessary if mention in demo
- Often when asking fro completion, youre in an incomplete state. We need a way of greeting semantic information in this case

Hazel LLM Assistant: Conversational Architecture



System Message: Basic Instructions

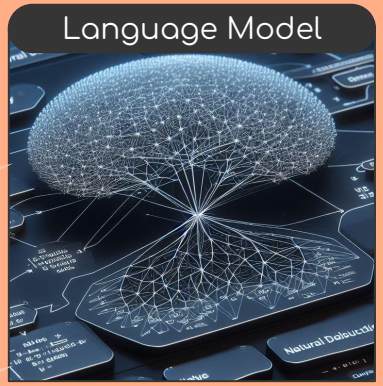
User Message: Program Sketch

Language Server: Type Information

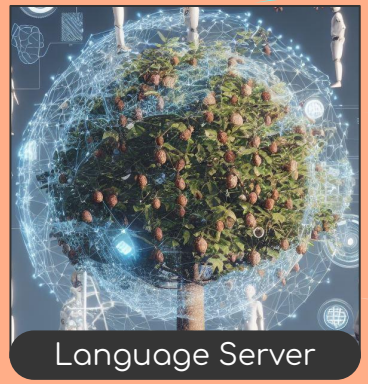
Assistant Message:
Suggested completion

Language Server: Type errors, if any

Assistant Message:
Corrected completion, if necessary



Hazel LLM Assistant: Conversational Architecture



System Message: Generic Instructions

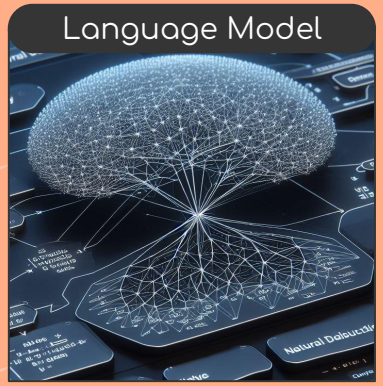
User Message: Program Sketch

Language Server: Type Information

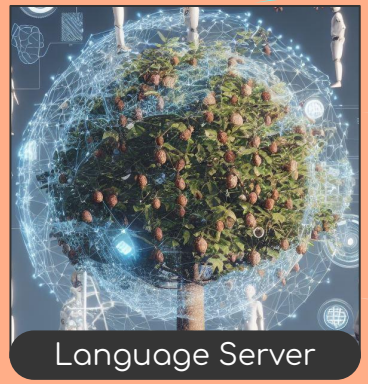
Assistant Message:
Suggested completion

Language Server: Type errors, if any

Assistant Message:
Corrected completion, if necessary



Hazel LLM Assistant: Conversational Architecture



System Message: Generic Instructions

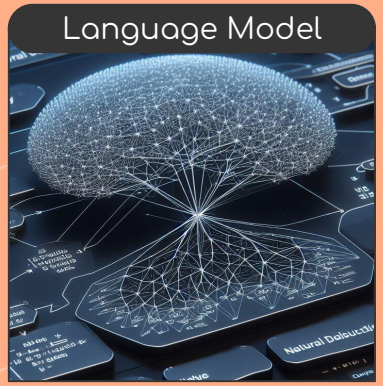
User Message: Program Sketch

Language Server: Type Information

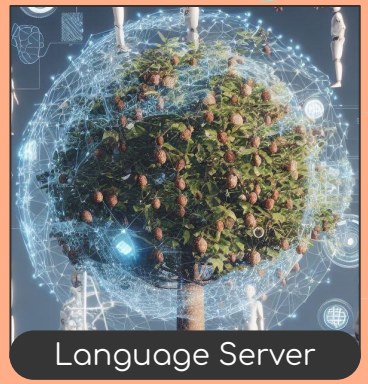
Assistant Message:
Suggested completion

Language Server: Type errors, if any

Assistant Message:
Corrected completion, if necessary



Hazel LLM Assistant: Conversational Architecture



System Message: Generic Instructions

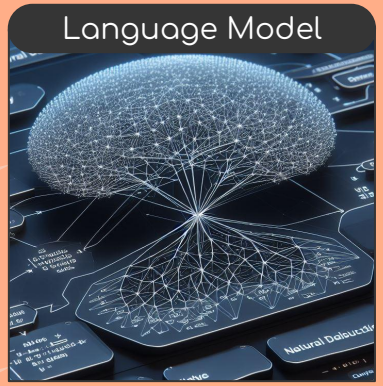
User Message: Program Sketch

Language Server: Type Information

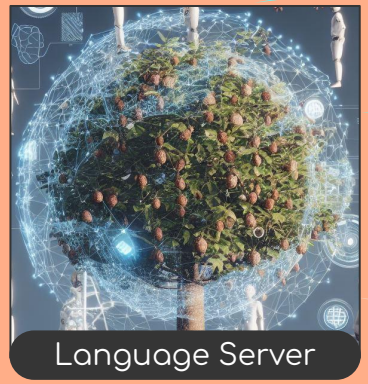
Assistant Message:
Suggested completion

Language Server: Type errors, if any

Assistant Message:
Corrected completion, if necessary



Hazel LLM Assistant: Conversational Architecture



System Message: Generic Instructions

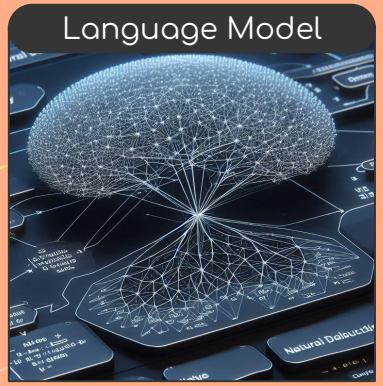
User Message: Program Sketch

Language Server: Type Information

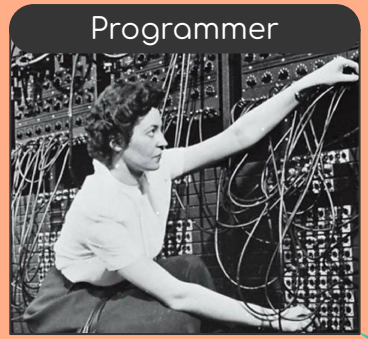
Assistant Message:
Suggested completion

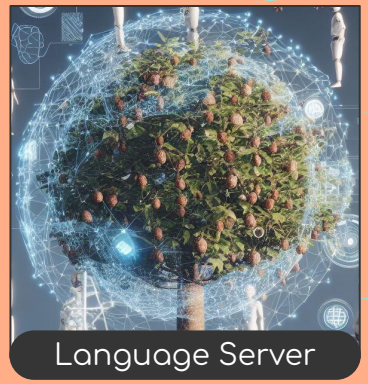
Language Server: Type errors, if any

Assistant Message:
Corrected completion, if necessary



Hazel LLM Assistant: Conversational Architecture





System Message: Generic Instructions

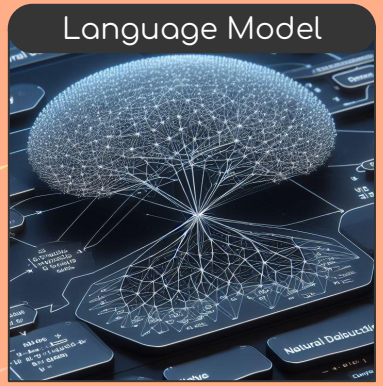
User Message: Program Sketch

Language Server: Type Information

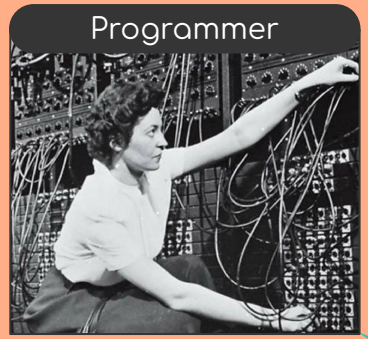
Assistant Message:
Suggested completion

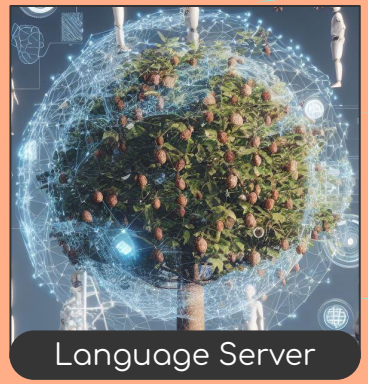
Language Server: Type errors, if any

Assistant Message:
Corrected completion, if necessary



Hazel LLM Assistant: Conversational Architecture





System Message: Generic Instructions

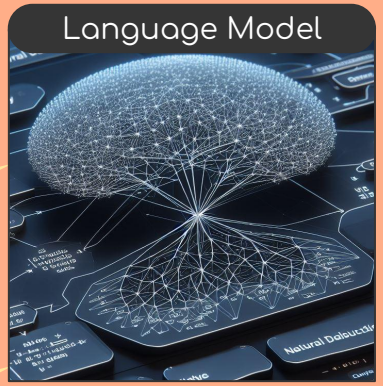
User Message: Program Sketch

Language Server: Type Information

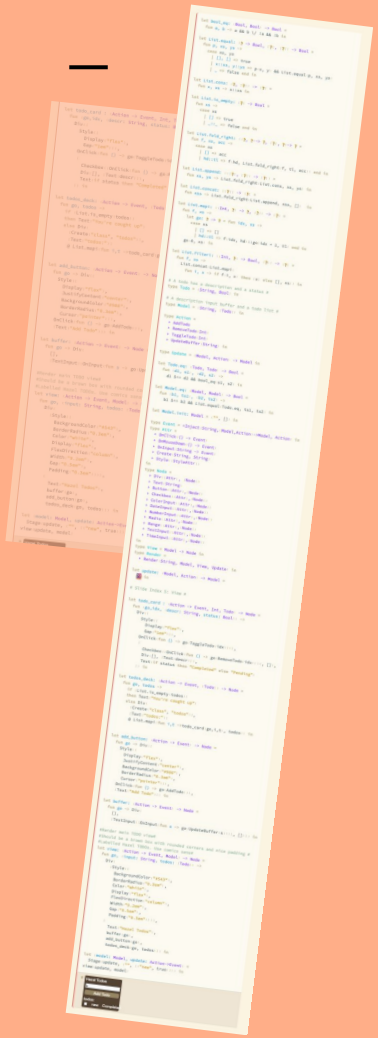
Assistant Message:
Suggested completion

Language Server: Type errors, if any

Assistant Message:
Corrected completion, if necessary

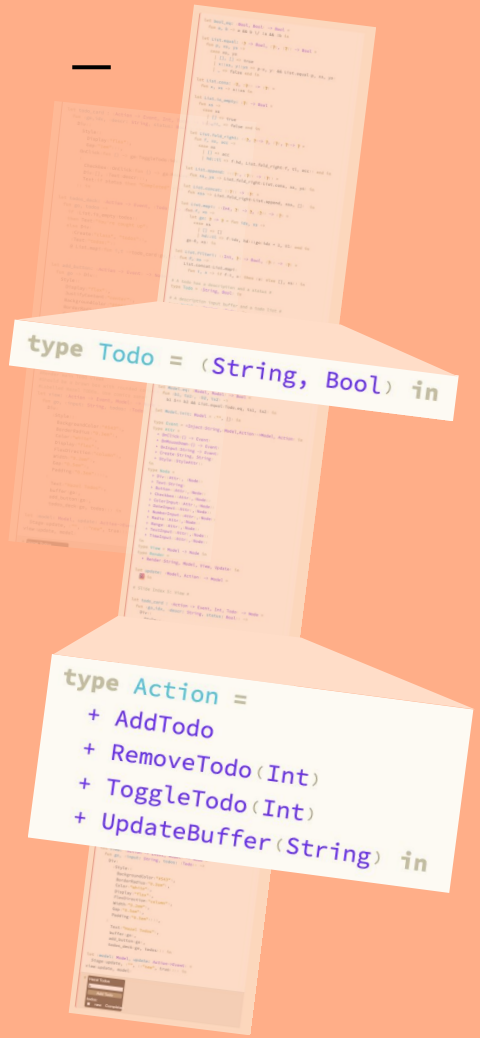


Prompt Construction



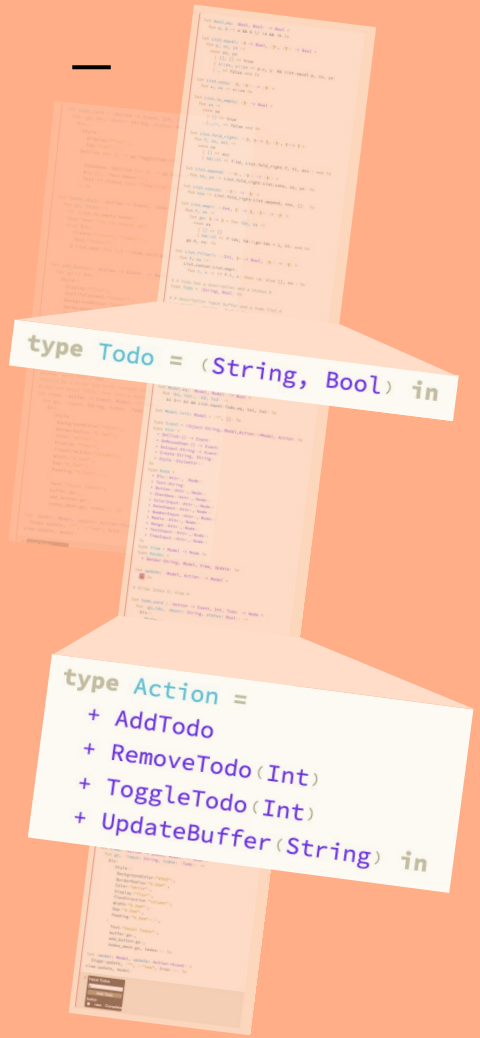
Prompt Construction: Challenges

- Limited context window length; can't just throw everything in
- Per-token costs
- Even with larger windows, models aren't equally attentive to the whole length



Prompt Construction: Challenges

- Limited context window length; can't just throw everything in
- Per-token costs
- Even with larger windows, models aren't equally attentive to the whole length
- **Need to distill pertinent data**

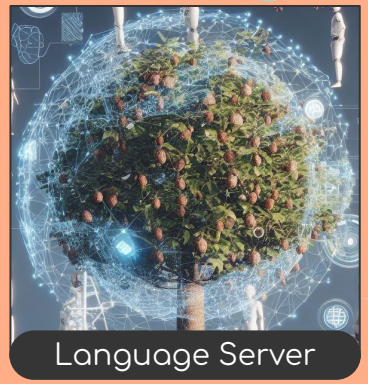


Prompt Construction: Approaches

- Basic approach:
  - Truncate program prefix to window length
- Current approaches: Heuristics for related code:
  - Last active tabs (Github Co-pilot)
  - Imports via dependency analysis (RepoCoder: Zhang et al. 2023)
  - RAG: Vector databases to retrieve ‘similar’ code

Type-Directed Prompt Construction

Hazel LLM Assistant: Conversational Architecture



System Message: Basic Instructions

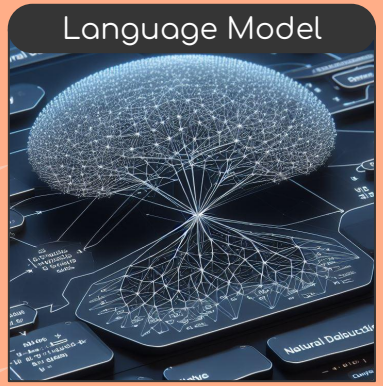
User Message: Program Sketch

Language Server: Type Information

Assistant Message:
Suggested completion

Language Server: Type errors, if any

Assistant Message:
Corrected completion, if necessary



```
list(string)
let main_prompt = [
  "CODE COMPLETION INSTRUCTIONS:"
  "- Reply with a functional, idi
  "- Reply only with a single rep
  "- Reply only with code",
  "- DO NOT suggest more replacem
  "- DO NOT include the program s
  "- DO NOT include a period at t
];
```

```
list(string)
let hazel_syntax_notes = [
  "HAZEL SYNTAX NOTES:",
  "- Hazel uses C-style function
  "- Function application is ALWA
  "- Function parameters are ALWA
  "- There is no dot accessor not
  "- The following ARE NOT Hazel
  "- Pattern matching is ALWAYS w
  "- The ONLY way to define a nam
  "- No 'rec' keyword is necessar
  "- Format the code with proper
];
```

```
 (
   "case Foo(5) | Foo(x) => ??
   Type.expected(Some(Ana(Int)
   "x",
 ),
 (
   "let num_or_zero = fun mayb
   Type.expected(Some(Syn), ~c
   "num",
 ),
 (
   "let merge_sort: [Int]->[In
   Type.expected(Some(Ana(Arro
   "fun list ->\nlet split: [I
 ),
 (
   "type MenuItem =\n+ Breakfa
   Type.expected(Some(Ana(Var(
   "fun m ->\ncase m\n| Breakf
 ),
```

Prompt Construction: Hazel Assistant

- System Message (Same for each prompt)
  - Task Description
  - Syntax Specification
  - Input/Output Examples (Few-shot)

Prompt Construction: Hazel Assistant

- System Message (Same for each prompt)
  - **Task Description**
  - Syntax Specification
  - Input/Output Examples (Few-shot)

```
list(string)
let main_prompt = [
  "CODE COMPLETION INSTRUCTIONS:",
  "- Reply with a functional, idiomatic replacement for the program hole marked '??' in the provided program sketch",
  "- Reply only with a single replacement term for the unqiue distinguished hole marked '??'",
  "- Reply only with code",
  "- DO NOT suggest more replacements for other holes in the sketch (marked '?'), or implicit holes",
  "- DO NOT include the program sketch in your reply",
  "- DO NOT include a period at the end of your response and DO NOT use markdown",
];
```

Prompt Construction: Hazel Assistant

- System Message (Same for each prompt)
  - Task Description
  - **Syntax Specification**
  - Input/Output Examples (Few-shot)

```
list(string)
let hazel_syntax_notes = [
  "HAZEL SYNTAX NOTES:",
  "- Hazel uses C-style function application syntax, with parenthesis around comma-separated arguments",
  "- Function application is ALWAYS written using parentheses and commas: use 'function(arg1, arg2)'. DO NOT just us
  "- Function parameters are ALWAYS commas separated: 'fun arg1, arg2 -> <exp>'. DO NOT use spaces to separate funct
  "- There is no dot accessor notation for tuples; DO NOT use tuple.field. use pattern matching for destructuring: 
  "- The following ARE NOT Hazel keywords. DO NOT use these keywords: switch, with, of, rec. ALWAYS omit these keywo
  "- Pattern matching is ALWAYS written a 'case ... end' expression. Cases MUST END in an 'end' keyword. DO NOT USE
  "- The ONLY way to define a named function is by using a function expression nested in a let expression like 'let
  "- No 'rec' keyword is necessary for 'let' to define a recursive function. DO NOT use the 'rec' keyword with 'let
  "- Format the code with proper linebreaks",
];
```

Prompt Construction: Hazel Assistant

- System Message (Same for each prompt)
  - Task Description
  - Syntax Specification
  - **Input/Output Examples (Few-shot)**

```
list(string)
let main_prompt = [
  "CODE COMPLETION INSTRUCTIONS:
  "- Reply with a functional, id
  "- Reply only with a single re
  "- Reply only with code",
  "- DO NOT suggest more replace
  "- DO NOT include the program
  "- DO NOT include a period at
];
```

```
list(string)
let hazel_syntax_notes = [
  "HAZEL SYNTAX NOTES:",
  "- Hazel uses C-style function
  "- Function application is ALWA
  "- Function parameters are ALWA
  "- There is no dot accessor not
  "- The following ARE NOT Hazel
  "- Pattern matching is ALWAYS w
  "- The ONLY way to define a nam
  "- No 'rec' keyword is necessar
  "- Format the code with proper
];
```

```
(
  "case Foo(5) | Foo(x) => ??
  Type.expected(Some(Ana(Int)
  "x",
),
(
  "let num_or_zero = fun mayb
  Type.expected(Some(Syn), ~c
  "num",
),
(
  "let merge_sort: [Int]->[In
  Type.expected(Some(Ana(Arro
  "fun list ->\nlet split: [I
),
(
  "type MenuItem =\n+ Breakfa
  Type.expected(Some(Ana(Var(
  "fun m ->\ncase m\n| Breakf
),
```

```
  "fun x:Int -> ??",
),
(
  "let get: Option_int => Int = " ++
  "  case Some(5)" ++
  "  | Some(x) => ??" ++
  "  | None => 0 end",
  Type.expected(Some(Ana(Int)), ~ctx=[]),
  "x",
),
(
  "let num_or_zero = fun maybe_num -> case maybe_num | Some(num) => ??
```

Prompt Construction: Hazel Assistant

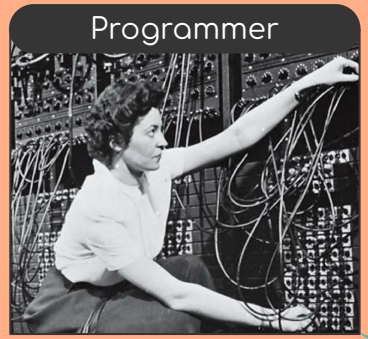
- System Message (Same for each prompt)
  - Task Description
  - Syntax Specification
  - **Input/Output Examples (Few-shot)**

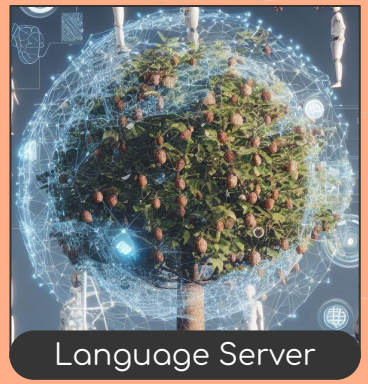
```
list(string)
let main_prompt = [
  "CODE COMPLETION INSTRUCTIONS:"
  "- Reply with a functional, idi
  "- Reply only with a single rep
  "- Reply only with code",
  "- DO NOT suggest more replacem
  "- DO NOT include the program s
  "- DO NOT include a period at t
];
```

```
list(string)
let hazel_syntax_notes = [
  "HAZEL SYNTAX NOTES:",
  "- Hazel uses C-style function
  "- Function application is ALWA
  "- Function parameters are ALWA
  "- There is no dot accessor not
  "- The following ARE NOT Hazel
  "- Pattern matching is ALWAYS w
  "- The ONLY way to define a nam
```

```
(
  "let num_or_zero = fun maybe_num -> case ma
  Type.expected(Some(Syn), ~ctx=[]),
  "num",
),
(
  "let merge_sort: [Int]->[Int] =\n??\nin\nme
  Type.expected(Some(Ana(Arrow(Int, Int))), ~
  "fun list ->\nlet split: [Int]->([Int],[Int
),
```

Hazel LLM Assistant: Conversational Architecture





System Message: Basic Instructions

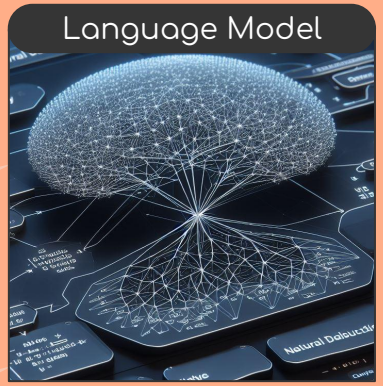
User Message: Program Sketch

Language Server: Type Information

Assistant Message:
Suggested completion

Language Server: Type errors, if any

Assistant Message:
Corrected completion, if necessary



Prompt Construction: Hazel Assistant

- Specific (Varies per prompt)
  - **Program Sketch (around Cursor)**
  - Expected Type (at Cursor)
  - Typing Context (at Cursor)

```
# Handle TODO updates from view layer #
let update: (Model, Action) -> Model =
  fun (input, todos), action ->
    ?? in
```

Prompt Construction: Hazel Assistant

- Specific (Varies per prompt)
  - **Program Sketch (around Cursor)**
  - Expected Type (at Cursor)
  - Typing Context (at Cursor)

```
# Handle TODO updates from view layer #
let update: (Model, Action) -> Model =
  fun (input, todos), action ->
    ?? in
```

Prompt Construction: Hazel Assistant

- Specific (Varies per prompt)
  - Program Sketch (around Cursor)
  - **Expected Type (at Cursor)**
  - Typing Context (at Cursor)

```
# Handle TODO updates from view layer #
let update: (Model, Action) -> Model =
  fun (input, todos), action ->
    ??
```

Γ EXP ? Empty expression hole

Fillable by any expression of type `Model`

Prompt Construction: Hazel Assistant

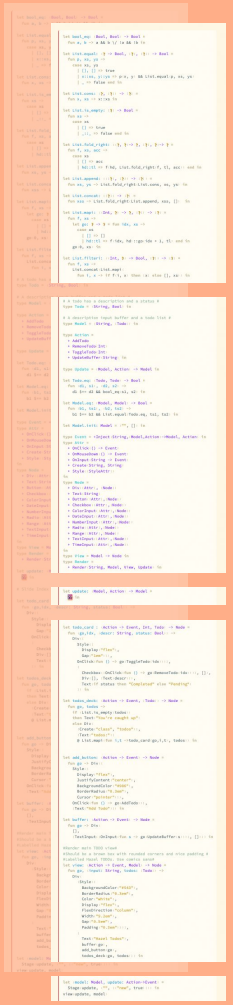
- Specific (Varies per prompt)
  - Program Sketch (around Cursor)
  - Expected Type (at Cursor)
  - **Typing Context (at Cursor)**

```
# Handle TODO updates from view layer
let update: (Model, Action
  fun (input
    ??
```

```
toggle : (Int, [Todo]) -> [Todo]
input : String
todos : [Todo]
action : Action
```

Γ EXP ? Empty expression hole

Fillable by any expression of type `Model`



Prompt Construction: Hazel Assistant

- Specific (Varies per prompt)
  - Program Sketch (around Cursor)
  - Expected Type (at Cursor)
  - **Typing Context (at Cursor)**

```
toggle : (Int, [Todo]) -> [Todo]
input : String
todos : [Todo]
action : Action
```

Γ EXP ? Empty expression hole

Fillable by any expression of type `Model`

Prompt Construction: Hazel Assistant

```
# A todo has a description and a status #
type Todo = (String, Bool) in

# A description input buffer and a todo list #
type Model = (String, [Todo]) in

type Action =
  + AddTodo
  + RemoveTodo(Int)
  + ToggleTodo(Int)
  + UpdateBuffer(String) in

type Update = (Model, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
  fun ((d1, s1), (d2, s2)) ->
    d1 $== d2 && bool_eq(s1, s2) in

let Model.eq: (Model, Model) -> Bool =
  fun ((b1, ts1), (b2, ts2)) ->
    b1 $== b2 && List.equal(Todo.eq, ts1, ts2) in

let Model.init: Model = ("", []) in

type Event = +Inject(String,(Model,Action)->Model, Action) in
type Attr =
  + OnClick(() -> Event)
  + OnMouseDown(() -> Event)
  + OnInput(String -> Event)
  + Create(String, String)
  + Style((StyleAttr))
in
type Node =
  + Div((Attr), [Node])
  + Text(String)
  + Button((Attr),(Node))
  + Checkbox((Attr),(Node))
  + ColorInput((Attr),(Node))
  + DateInput((Attr),(Node))
  + NumberInput((Attr),(Node))
  + Radio((Attr),(Node))
  + Range((Attr),(Node))
  + TextInput((Attr),(Node))
  + TimeInput((Attr),(Node))
in
type View = Model -> Node in
type Render =
  + Render(String, Model, View, Update) in
```

```
toggle : (Int, [Todo]) -> [Todo]
input : String
todos : [Todo]
action : Action
```

Γ EXP ? Empty expression hole

Fillable by any expression of type `Model`

Prompt Construction: Hazel Assistant

```
type Action =
  + AddTodo
  + RemoveTodo(Int)
  + ToggleTodo(Int)
  + UpdateBuffer(String) in
```

```
type Model = (String, [Todo]) in
```

```
toggle : (Int, [Todo]) -> [Todo]
input : String
todos : [Todo]
action : Action
```

Γ EXP ? Empty expression hole

Fillable by any expression of type `Model`

Prompt Construction: Hazel Assistant

```
type Action =
  + AddTodo
  + RemoveTodo(Int)
  + ToggleTodo(Int)
  + UpdateBuffer(String) in
```

```
type Todo = (String, Bool) in
```

```
type Model = (String, [Todo]) in
```

```
toggle : (Int, [Todo]) -> [Todo]
input : String
todos : [Todo]
action : Action
```

Γ EXP ? Empty expression hole

Fillable by any expression of type `Model`

Prompt Construction: Hazel Assistant

```
# Handle TODO updates from view layer #
let update: (Model, Action) -> Model =
  fun (input, todos), action ->
    ?? in
```

```
type Action =
  + AddTodo
  + RemoveTodo(Int)
  + ToggleTodo(Int)
  + UpdateBuffer(String) in
```

```
type Todo = (String, Bool) in
```

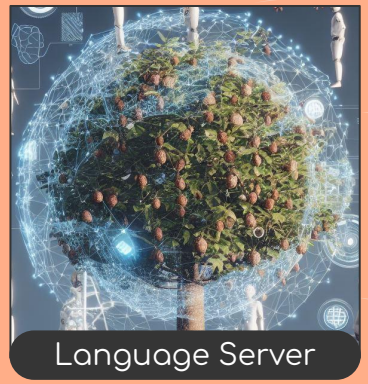
```
type Model = (String, [Todo]) in
```

```
toggle : (Int, [Todo]) -> [Todo]
input : String
todos : [Todo]
action : Action
```

Γ EXP ? Empty expression hole

Fillable by any expression of type `Model`

Hazel LLM Assistant: Conversational Architecture



System Message: Basic Instructions

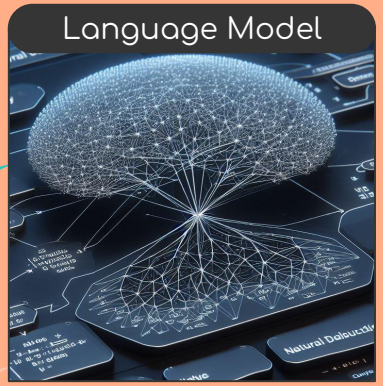
User Message: Program Sketch

Language Server: Type Information

Assistant Message:
Suggested completion

Language Server: Type errors, if any

Assistant Message:
Corrected completion, if necessary



TODO

- completion

Error Correction

Error correction round

- The language server intercepts responses and sends any errors back to the LLM

- Interestingly, this provided us with impetus to write better, more human-readible error messages for hazel!
- We only do one round of error correction so still could be errors
- TODO: Show example of it getting it wrong

Preliminary Results

Experiment targets

- Type-augmented few-shot sketch completion
- Evaluation against held-out tests
- Todo: Difficulty of evaluation

Is this effective?

- It can be.
- Especially in the case where the types are large ADTs not appearing in the same file as the sketch
- Note: This is an effective technique in copilot for eg OCaml (copy/paste relevant definitions to before a completion)

Inclusion of type information

- Forthcoming
- Very effective if types reference large ADTs not defined in the same file
- Include examples with and without
- Basic figures

Generic prompt efficacy

- Inclusion of syntax description doesn't make that much of a difference
- Neither do instructions (usually)
- Examples however are very load-bearing
- Graph forthcoming

Forthcoming tests

- More diverse examples
- If we want to move to a more established benchmark, we'll likely have to move beyond Hazel
- Have been considering OCaml and Typescript as possible targets, though these make gathering granular type information more difficult

TODO Example

Update.haze

hazel Scratch 5 / 11

```
# Handle TODO updates from view layer #
let update: (Model, Action) -> Model =
  ⟨⟩ in

EXPORT
```

Hazel TODOs App

Util.haze

```
let bool_eq: (Bool, Bool) -> Bool =
  fun a, b -> a && b \/ !a && !b in

let List.equal: (? -> Bool, [?], [?]) -> Bool =
  fun p, xs, ys ->
    case xs, ys
      | [], [] => true
      | x::xs, y::ys => p(x, y) && List.equal(p, xs, ys)
      | _ => false end in

let List.cons: (?, [?]) -> [?] =
  fun x, xs -> x::xs in

let List.is_empty: [?] -> Bool =
  fun xs ->
    case xs
      | [] => true
      | _::_ => false end in

let List.fold_right: (?, ?) -> ?, [?], ?) -> ? =
  fun f, xs, acc ->
    case xs
      | [] => acc
      | hd::tl => f(hd, List.fold_right(f, tl, acc)) end in

let List.append: ([?], [?]) -> [?] =
  fun xs, ys -> List.fold_right(List.cons, xs, ys) in

let List.concat: ([?]) -> [?] =
  fun xss -> List.fold_right(List.append, xss, []) in

let List.map1: (Int, ?) -> ?, [?]) -> [?] =
  fun f, xs ->
    let go: ? -> ? = fun idx, xs ->
      case xs
        | [] => []
        | hd::tl => f(idx, hd)::go(idx + 1, tl) end in
    go(0, xs) in

let List.filter1: (Int, ?) -> Bool, [?]) -> [?] =
  fun f, xs ->
    List.concat(List.map1(
      fun i, x -> if f(i, x) then [x] else [], xs)) in
```

Model.haze

```
# A todo has a description and a status #
type Todo = (String, Bool) in

# A description input buffer and a todo list #
type Model = (String, [Todo]) in

type Action =
  + AddTodo
  + RemoveTodo(Int)
  + ToggleTodo(Int)
  + UpdateBuffer(String) in

type Update = (Model, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
  fun (d1, s1), (d2, s2) ->
    d1 $== d2 && bool_eq(s1, s2) in

let Model.eq: (Model, Model) -> Bool =
  fun (b1, ts1), (b2, ts2) ->
    b1 $== b2 && List.equal(Todo.eq, ts1, ts2) in

let Model.init: Model = ("", []) in

type Event = +Inject(String, Model,Action)->Model, Action) in
type Attr =
  + OnClick(() -> Event)
  + OnMouseDown(() -> Event)
  + OnInput(String -> Event)
  + Create(String, String)
  + Style([StyleAttr])
in
type Node =
  + Div([Attr], [Node])
  + Text(String)
  + Button([Attr],[Node])
  + Checkbox([Attr],[Node])
  + ColorInput([Attr],[Node])
  + DateInput([Attr],[Node])
  + NumberInput([Attr],[Node])
  + Radio([Attr],[Node])
  + Range([Attr],[Node])
  + TextInput([Attr],[Node])
  + TimeInput([Attr],[Node])
in
type View = Model -> Node in
type Render =
  + Render(String, Model, View, Update) in
```

Update.haze

```
let update: (Model, Action) -> Model =
  in
```

View.haze

```
let todo_card : (Action -> Event, Int, Todo) -> Node =
  fun (go,idx, (descr: String, status: Bool)) ->
    Div([
      Style([
        Display("flex"),
        Gap("1em")]),
      OnClick(fun () -> go(ToggleTodo(idx))]),
      [
        Checkbox([OnClick(fun () -> go(RemoveTodo(idx))]), []),
        Div([], [Text(descr)]),
        Text(if status then "Completed" else "Pending")
        ]) in

let todos_deck: (Action -> Event, [Todo]) -> Node =
  fun go, todos ->
    if (List.is_empty(todos))
    then Text("You're caught up")
    else Div([
      Create("class", "todos")],
      [Text("todos")],
      @ List.map1(fun i,t ->todo_card(go,i,t), todos)) in


let add_button: (Action -> Event) -> Node =
  fun go -> Div([
    Style([
      Display("flex"),
      JustifyContent("center"),
      BackgroundColor("#9B9"),
      BorderRadius("0.3em"),
      Cursor("pointer")]),
    OnClick(fun () -> go(AddTodo)]),
    [Text("Add Todo")]]) in

let buffer: (Action -> Event) -> Node =
  fun go -> Div([
    [],
    [TextInput([OnInput(fun s -> go(UpdateBuffer(s))]), []])]) in

#Render main TODO view#
#Should be a brown box with rounded corners and nice padding #
#Labelled Hazel TODOs. Use comics sansa
let view: (Action -> Event, Model) -> Node =
  fun go, (input: String, todos: [Todo]) ->
    Div([
      [Style([
        BackgroundColor("#543"),
        BorderRadius("0.3em"),
        Color("white"),
        Display("flex"),
        FlexDirection("column"),
        Width("5.2em"),
        Gap("0.5em"),
        Padding("0.3em")])],
        [
        Text("Hazel Todos"),
        buffer(go),
        add_button(go),
        todos_deck(go, todos)]) in
```

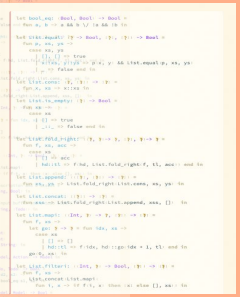
Main.haze

```
let (model: Model, update: Action->Event) =
  Stage(update, ("", ([("new", true)])]) in
view(update, model)
```

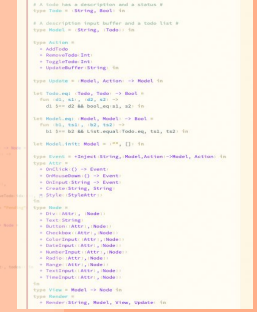
Prompt Construction: Hazel Assistant

- Specific
  - **Program Sketch (around Cursor)**
  - Expected Type (at Cursor)
  - Typing Context (at Cursor)

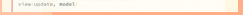
Util.haze



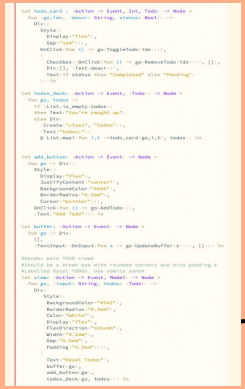
Model.haze



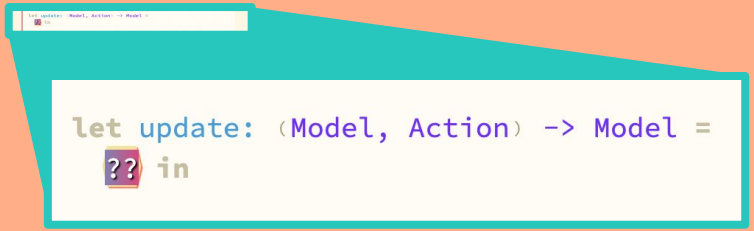
Main.haze



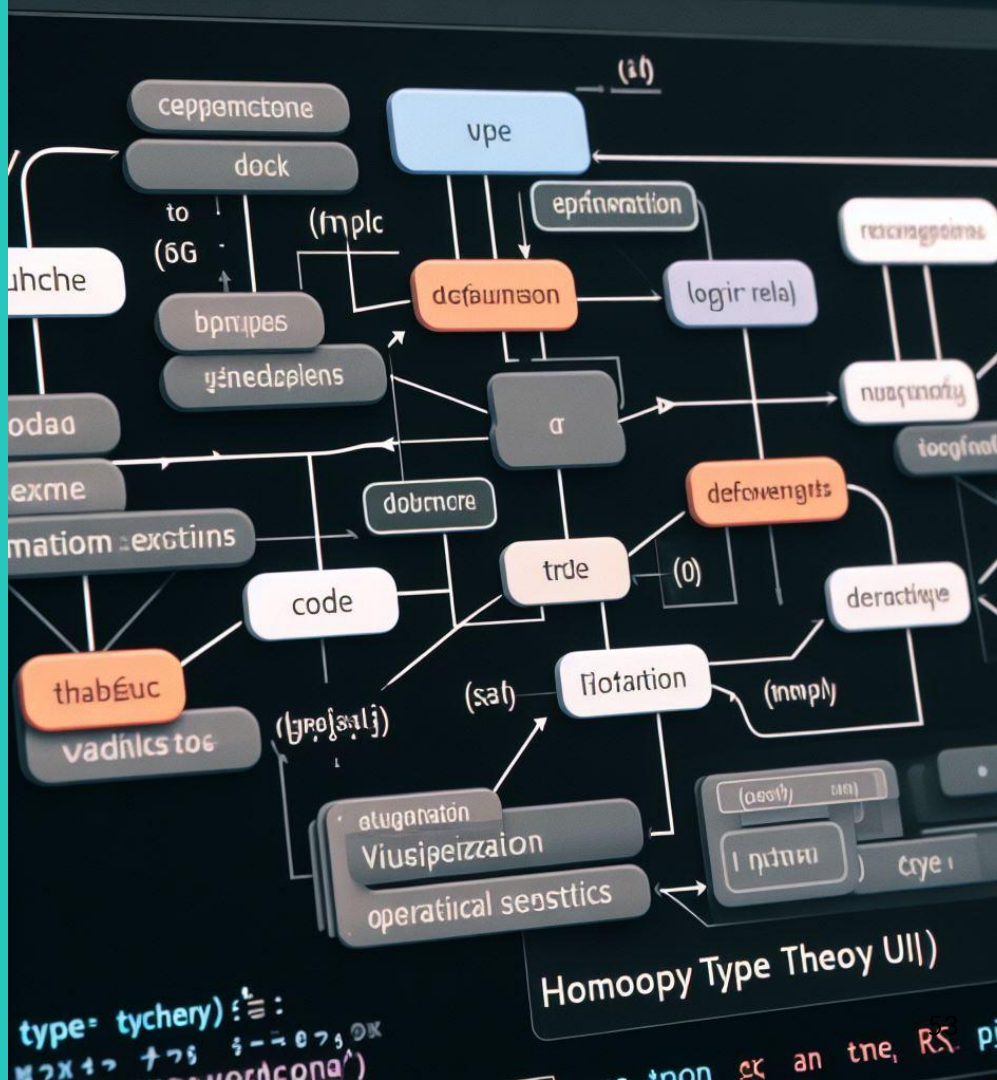
View.haze



Update.haze



Construction Time Type Correctness



Construction Time Type Correctness

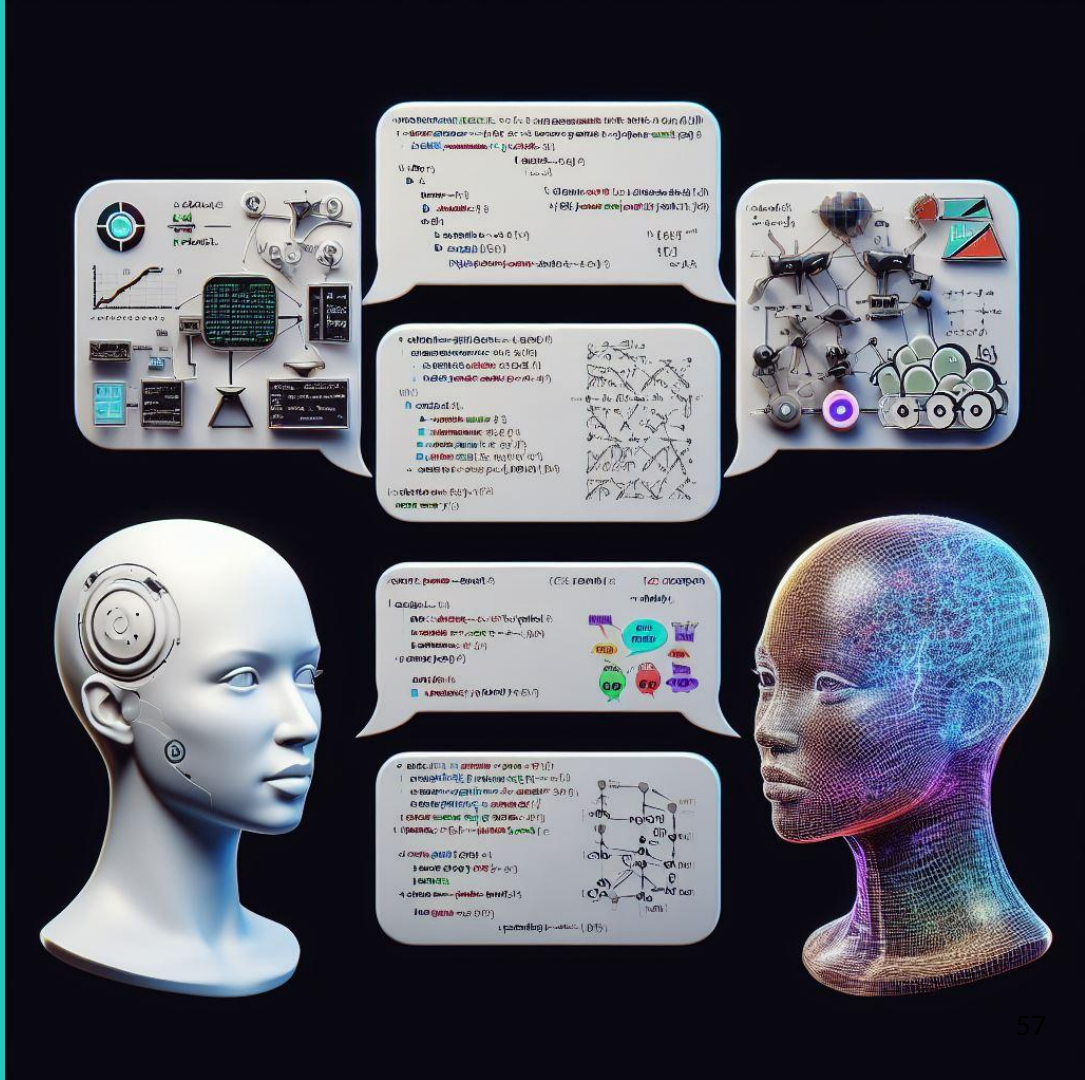
- Token masking
- CFGs in llama 2
- MSR paper
- REPILOT
- Why we are using hazel: language services

Dynamic Information

Dynamic Information

- Dynamic data with Smyth

Strategies & Collaboration



Forthcoming

- Forthcoming
- LET IT TALK TO LANGAUGE SERVER AND ASK FOR WHAT IT WANTS
- TOOLFORMER
- Bring back dialogue photo

Dialogue with Language Server

- More

Takeaways

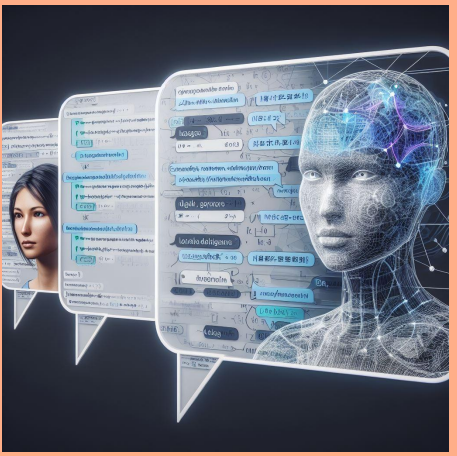
- Possible with prompt engineering to get good LLM completion in at least one niche languages
- Type information provides an effective guide to collate program sketches in at least some situations
- THERE IS SO MUCH TODO

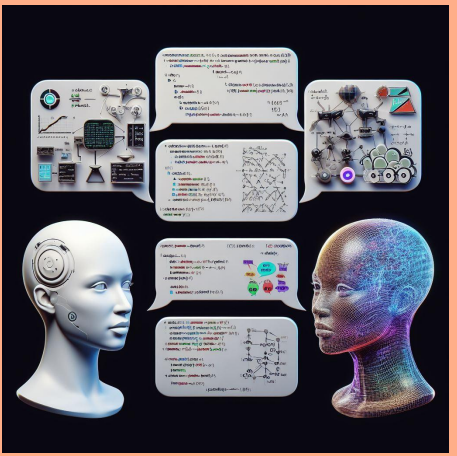
Thanks for listening!

Questions?

TODOS

- Citations (throughout):
  - Repilot
  - Msr type paper
- Results (lol)





More broadly:
3-way collab between User, LS, and LLM

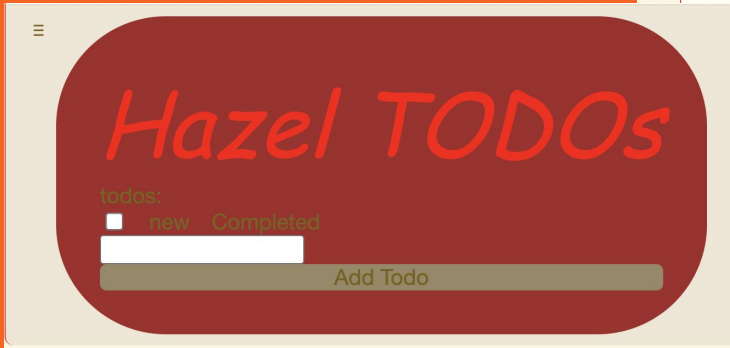
Type-Directed Prompt Construction:

- Type information provides a natural abstraction on program structure
- Type information enables linking local requirements to global resources
- Type information is thus a natural candidate to guide projecting a codebase into a limited context window informed by a particular point of interest; the completion site

In-progress modifications

- Replace fixed examples with type-specific ones when available
- More granular approach to including type information
- When completing arrow types, filter context for arrow types which take input or produce output
- Generally move to integrating lightweight program synthesis

Demo



hazel Examples Void

```
#Render main TODO view#
#Should be a brown box with rounded corners and nice padding #
#Labelled Hazel TODOs. Use comics sans#
let view: (Action -> Event, Model) -> Node =
  fun go, (buffer, todos) ->
    Div(
      [
        Style([
          BackgroundColor("#946"),
          BorderRadius("0.3em"),
          Padding("1em")]),
        Create("class", "app")
      ],
      [
        Div([
          Style([
            FontFamily("'Comic Sans MS', cursive, sans-serif"),
            Color("#fff"),
            Margin("1em"),
            Display("flex"),
            JustifyContent("center")]),
          Create("class", "header")
        ], [Text("Hazel TODOs")]),
        buffer(go),
        add_button(go),
        todos_deck(go,todos)
      ]
    ) in
```

Γ EXP ? Function literal : (Action -> Event, Model) -> Node equals expected type

Example

Update.haze

hazel Scratch 5 / 11

```
# Handle TODO updates from view layer #
let update: (Model, Action) -> Model =
  ?? in
```

Context

Model.haze

```
# A todo has a description and a status #
type Todo = (String, Bool) in

# A description input buffer and a todo list #
type Model = (String, [Todo]) in

type Action =
  + AddTodo
  + RemoveTodo(Int)
  + ToggleTodo(Int)
  + UpdateBuffer(String) in

type Update = (Model, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
  fun (d1, s1), (d2, s2) ->
    d1 $== d2 && bool_eq s1, s2) in

let Model.eq: (Model, Model) -> Bool =
  fun (b1, ts1), (b2, ts2) ->
    b1 $== b2 && List.equal(Todo.eq, ts1, ts2) in

let Model.init: Model = ("", []) in

type Event = +Inject(String, Model,Action)->Model, Action) in
type Attr =
  + OnClick(() -> Event)
  + OnMouseDown(() -> Event)
  + OnInput(String -> Event)
  + Create(String, String)
  + Style(StyleAttr)
in
type Node =
  + Div (Attr , Node)
  + Text(String)
  + Button (Attr ,Node)
  + Checkbox (Attr ,Node)
  + ColorInput (Attr ,Node)
  + DateInput (Attr ,Node)
  + NumberInput (Attr ,Node)
  + Radio (Attr ,Node)
  + Range (Attr ,Node)
  + TextInput (Attr ,Node)
  + TimeInput (Attr ,Node)
in
type View = Model -> Node in
type Render =
  + Render(String, Model, View, Update) in
```

Update.haze

```
let update: (Model, Action) -> Model =
  in
```

—

Model.haze

```
# A todo has a description and a status #
type Todo = (String, Bool) in

# A description input buffer and a todo list #
type Model = (String, [Todo]) in

type Action =
  + AddTodo
  + RemoveTodo(Int)
  + ToggleTodo(Int)
  + UpdateBuffer(String) in

type Update = (Model, Action) -> Model in

let Todo.eq: (Todo, Todo) -> Bool =
  fun ((d1, s1), (d2, s2)) ->
    d1 $== d2 && bool_eq(s1, s2) in

let Model.eq: (Model, Model) -> Bool =
  fun ((b1, ts1), (b2, ts2)) ->
    b1 $== b2 && List.equal(Todo.eq, ts1, ts2) in

let Model.init: Model = ("", []) in

type Event = +Inject(String,(Model,Action)->Model, Action) in
type Attr =
  + OnClick(() -> Event)
  + OnMouseDown(() -> Event)
  + OnInput(String -> Event)
  + Create(String, String)
  + Style((StyleAttr))
in
type Node =
  + Div((Attr), [Node])
  + Text(String)
  + Button((Attr),[Node])
  + Checkbox((Attr),[Node])
  + ColorInput((Attr),[Node])
  + DateInput((Attr),[Node])
  + NumberInput((Attr),[Node])
  + Radio((Attr),[Node])
  + Range((Attr),[Node])
  + TextInput((Attr),[Node])
  + TimeInput((Attr),[Node])
in
type View = Model -> Node in
type Render =
  + Render(String, Model, View, Update) in
```

Update.haze

hazel Scratch 5 / 11

```
# Handle TODO updates from view layer #
let update: (Model, Action) -> Model =
  ?? in
```

```
Checkbox : ([Attr], [Rec α.
Button : ([Attr], [Rec α. Di
Text : String -> Node
Div : ([Attr], [Rec α. Div((
View :: Model -> Node
Render :: +Render((String, M
Render : (String, Model, Vie
```

Γ EXP ? Empty expression hole Fillable by any expression of type (Model, Action) -> Model

Model.haze

```
type Todo = (String, Bool) in

type Model = (String, [Todo]) in

type View = Model -> Node in
type Render =
  + Render(String, Model, View, Update) in
```

Update.haze

hazel Scratch 5 / 11

```
# Handle TODO updates from view layer #
let update: (Model, Action) -> Model =
  ?? in
```

```
Checkbox : ([Attr], [Rec α.
Button : ([Attr], [Rec α. Di
Text : String -> Node
Div : ([Attr], [Rec α. Div((
View :: Model -> Node
Render :: +Render((String, M
Render : (String, Model, Vie
```

Γ EXP ? Empty expression hole Fillable by any expression of type (Model, Action) -> Model

Model.haze

```
type Model = (String, [Todo]) in
```

```
type View = Model -> Node in
type Render =
  + Render(String, Model, View, Update) in
```

Update.haze

hazel Scratch 5 / 11

```
# Handle TODO updates from view layer #
let update: (Model, Action) -> Model =
  ?? in
```

```
Checkbox : ([Attr], [Rec α.
Button : ([Attr], [Rec α. Di
Text : String -> Node
Div : ([Attr], [Rec α. Div((
View :: Model -> Node
Render :: +Render((String, M
Render : (String, Model, Vie
```

Γ EXP ? Empty expression hole Fillable by any expression of type (Model, Action) -> Model

Model.haze

```
type Todo = (String, Bool) in

type Model = (String, [Todo]) in

type Action =
  + AddTodo
  + RemoveTodo(Int)
  + ToggleTodo(Int)
  + UpdateBuffer(String) in

type View = Model -> Node in
type Render =
  + Render(String, Model, View, Update) in
```

Update.haze

hazel Scratch 5 / 11

```
# Handle TODO updates from view layer #
let update: (Model, Action) -> Model =
  ?? in
```

```
Checkbox : ([Attr], [Rec α.
Button : ([Attr], [Rec α. Di
Text : String -> Node
Div : ([Attr], [Rec α. Div((
View :: Model -> Node
Render :: +Render((String, M
Render : (String, Model, Vie
```

Γ EXP ? Empty expression hole Fillable by any expression of type (Model, Action) -> Model

Example

Update.haze

hazel Scratch 5 / 11

```
# Handle TODO updates from view layer #
let update: (Model, Action) -> Model =
  fun model, action ->
    let (buffer, todos) = model in
    case action
      | AddTodo =>
      if (buffer!= "") then ( "", (buffer, false)::todos)
      else model
        | RemoveTodo(index) => (buffer, List.filteri(fun idx, _ -> idx!= index, todos))
        | ToggleTodo(index) =>
        (buffer, List.mapi(fun idx, (t, b) -> if idx == index then (t, not(b)) else (t, b), todo
        | UpdateBuffer(new_buffer) => (new_buffer, todos)
    end in
```

Vague Outline

- demo with TODO example?
- LLM intro
- Prompt construction
- Prompt construction + types
- Some results
- Current work
- Future work

Intro

- Here today to talk about Type-directed Prompt Construction for LLM-powered Programming Assistants
- But first, something completely different

Demo

- Hazel MVU aka elm arch framework for live GUI dev
- Scroll thru TODO code highlighting model, update
- View fn, stubbed out
- Briefly start via type-directed autocomplete
  (Only start div? Or start attrs as well?)
- Delete that, write comment, and the llm-complete

Abstract for reference

- Large language models have changed the landscape for code completion, but the techniques by which prompts are constructed and model responses are handled do not fully leverage our available wealth of semantic information. We are investigating using semantic analysis to direct the collation and annotation of partial program sketches fed to the LLM to inform completions. In particular, we use bidirectional type inference in the Hazel programming environment to selectively supply types and definitions deemed relevant given the lexical context. The model's responses are then iteratively refined via dialog with a language server. We will present some early results of experiments on the potential of type-augmented construction, specifically within the domain of live GUI programming in Hazel. We will also describe planned experiments, and touch on ongoing efforts to extend the system, including: using dynamic analysis, semantically constraining construction at a per-token level, and looking beyond completion towards more strategic programming tasks. Finally, we report on the current viability of LLMs for code completion for a niche language not represented in the model training data.