顯式控制與自動推導：設計者應要求使用者說多少，又替使用者猜多少？

Explicit Control and Automatic Inference: How Much Should Programmers State, and How Much Should Languages Derive?

v1.0 / 2026-07-30 / Neo.K / 公開版／第二部核心風格原型正式論文

https://thisoneisneok.com/html/papers/pldst-007.html

英文名稱： Explicit Control and Automatic Inference: How Much Should Programmers State, and How Much Should Languages Derive?

系列： Programming Language Designer Style Taxonomy（PLDST）

文件編號： PLDST-007

版本： v1.0

日期： 2026-07-30

作者： Neo.K

文件狀態： 公開版／第二部核心風格原型正式論文

SECTION

摘要

程式語言設計長期在兩種願望之間拉扯：

- 使用者應明確寫出型別、資源、控制、依賴與意圖；
- 編譯器應利用上下文，自動推導可由機器確定的資訊。

過度顯式可能造成樣板、視覺噪音、重複契約與維護負擔；過度隱式則可能造成非局部搜尋、意外轉換、推導不穩定、難以解釋的錯誤，以及公開 API 隨實作細節漂移。

Milner 的多型型別理論與 Damas–Milner 的 principal type-scheme 結果，展示了在受控語言核心中，自動推導可以產生具有明確數學地位的最一般型別，而非任意猜測；Haskell 2010 將未提供型別簽章的繫結賦予對應的 inferred／principal type，但也保留型別簽章、類別約束、預設化與 Monomorphism Restriction 等邊界；Go 規格允許根據引數與限制關係推導泛型型別引數，但推導失敗時要求顯式補充，而不是任意選擇；Rust 允許區域型別與部分生命週期省略，但對函式與資料結構邊界保留明確規則；Kotlin 對區域變數、Builder 與泛型提供強推導，同時以 Explicit API mode 要求公共函式與屬性明確說明型別，避免實作改動意外改變 API；Scala 3 將 Scala 2 的廣泛 implicits 重構成 given、using、context parameter 與明確的 Conversion，其官方目標之一正是限制令人意外的隱式行為；TypeScript 則利用結構型別與 contextual typing，從變數位置、函式參數與回傳脈絡進行雙向推導。[R1][R2][R3][R4][R5][R6][R7][R8]

本文提出 顯式—推導配置模型（Explicitness–Inference Allocation Model, EIAM），將一項自動推導機制表示為：

> cal-I
> =
> (
> K,
> C,
> S,
> U,
> A,
> X,
> R,
> B
>)

其中：

- K：Known facts，既知事實；
- C：Constraints，約束；
- S：Search scope，搜尋範圍；
- U：Uniqueness，解是否唯一；
- A：Ambiguity policy，歧義政策；

- X：Explainability，可解釋性；
- R：Refactoring stability，重構穩定性；
- B：Boundary exposure，是否影響公開邊界。

本文區分七種經常被統稱為「隱式」的機制：

- 表面省略；
- 區域型別推導；
- 期望型別與雙向推導；
- 泛型、生命週期與效果約束求解；
- 上下文參數與證據合成；
- 隱式轉換與預設化；
- 公開契約推導。

它們具有完全不同的風險。局部變數型別推導通常只消除重複資訊；上下文證據合成則可能跨作用域搜尋；隱式轉換會改變程式的實際操作；公開契約推導更可能把私人實作細節永久化為外部承諾。

本文核心命題為：

> 【可推導
> not⇒
> 應省略】

以及：

> 【顯式
> not⇒
> 可理解】

成熟設計不應追求最大顯式或最大推導，而應遵守：

> 局部、唯一、穩定、可解釋且不形成外部契約的資訊，可以優先推導；跨模組、涉及能力、可能改變控制流、具有安全效果或會固定公共契約的資訊，應提高顯式程度。

關鍵詞： 程式語言設計、型別推導、顯式性、上下文推導、隱式參數、隱式轉換、生命週期省略、公開 API、PLDST

SECTION

一、顯式不等於冗長

顯式可以指：

- 寫出型別；
- 寫出型別引數；
- 寫出生命週期；
- 寫出效果；
- 寫出依賴；
- 寫出轉換；
- 寫出資源所有權；
- 寫出錯誤傳播；
- 寫出控制流；
- 寫出公開契約。

一段程式即使字元很多，也可能仍然隱藏：

- 動態查找；
- 全域狀態；
- 隱式 I/O；
- 隱式重試；

- 例外展開；
- 自動轉換；
- Context dependency。

因此：

> Length(program)
> not⇒
> Explicitness(program)

SECTION

二、推導不等於猜測

嚴格的型別推導通常是：

> KnownFacts+Constraints
> →
> UniqueSolution

而非：

> CompilerPreference
> →
> ConvenientGuess

Milner 的 Algorithm W 與 Damas–Milner principal type 結果之所以重要，是因為它們在特定語言與型別系統範圍內，能為可型別化表達式推導最一般型別。[R1][R2]

相反地，若系統：

- 在多個合法答案中任選；
- 依匯入順序改變結果；
- 依不可見全域狀態決定；

- 以啟發式選擇轉換；
- 版本更新後靜默選擇不同重載；

  則它更接近隱式決策，而不只是邏輯推導。

SECTION

三、省略不等於推導

例如：

fn f(x: &str) -> &str
Rust 的 lifetime elision 可能依固定規則補入生命週期。這是一種規則化省略。[R5]

而：

let x = expression
由使用方式與約束決定 x 的型別，屬於型別推導。

兩者都讓使用者少寫內容，但演算法、作用域與歧義邊界不同。

SECTION

四、推導不等於自動轉換

型別推導回答：

> 此表達式已經是什麼型別？

隱式轉換回答：

> 是否自動插入一個新操作，使它變成另一型別？

前者主要補全資訊；後者改變執行語義。

因此：

> Inference
> ≠
> Coercion
> ≠

把三者都稱為便利語法，會低估隱式轉換的風險。

SECTION

五、表面省略

表面省略以固定規則移除可預測標註，例如：

- Rust lifetime elision；
- 省略型別引數的預設語法；
- Context bound 展開；
- 省略 return；
- 自動 semicolon insertion。

判斷重點：

- 規則是否局部；
- 是否只有一種展開；
- 工具能否顯示展開；
- 省略是否改變公開契約。

SECTION

六、區域型別推導

典型形式：

```
let x = 42
val x = expression
var x = expression
```

其優點：

- 移除與右側重複的型別；
- 改變實作時不需同步修改局部標註；

- 讓複雜泛型型別不必人工重寫；
- 降低視覺噪音。

主要風險：

- 讀者離開 IDE 後看不到型別；
- 推導結果可能比預期更具體或更寬；
- 重構右側可能改變後續重載與 API；
- 空容器、數值字面值等可能缺少足夠資訊。

Rust Reference 將 _ 定義為請編譯器根據周圍資訊推導型別；若資訊不足，應報錯而不是產生動態型別。[R5]

SECTION

七、期望型別與雙向推導

推導可以由表達式向外，也可以由上下文向內。

TypeScript contextual typing 的例子是：函式運算式的參數型別可由它被指派的位置決定；泛型函式中，其他引數與期望回傳型別也可反向提供約束。[R8]

Kotlin Builder inference 則可利用 Lambda 內部對 Builder receiver 的呼叫，反向推導泛型型別引數。[R6]

優勢：

- Lambda 與 Callback 更精簡；
- DSL 可保持靜態型別；
- 泛型呼叫少寫型別引數；
- API 能把知識傳入區域表達式。

風險：

- 表達式離開原上下文後失去型別；
- 小型重構可能讓推導失敗；

- 錯誤位置遠離真正約束來源；
- 推導順序與敏感表達式可能影響診斷。

SECTION

八、泛型約束求解

Go 規格描述的型別推導，是從函式引數與參數間的 assignability 關係，以及型別參數限制之間求得型別引數。[R4]

這類推導的良好邊界是：

- 只從明確語言關係收集約束；
- 有唯一可接受結果；
- 不足時要求使用者補充型別引數；
- 不將偶然實作細節變成遠端搜尋。

泛型推導的目標不是取消泛型契約，而是取消呼叫點重複資訊。

SECTION

九、生命週期、效果與資源推導

Rust lifetime elision 允許在若干常見函式簽章中省略生命週期，但不是任意推導所有資料結構與 API 的生命週期。[R5]

更廣義的推導可以涵蓋：

- Effect；
- Region；
- Borrow；
- Capability；
- Nullability flow；

- Definite assignment；
- Ownership move。

這些資訊涉及安全與控制，因此推導系統需要：

- 明確失敗；
- 可解釋關係；
- 局部邊界；
- 逃生口標記；
- 不因模糊猜測降低保證。

SECTION

十、上下文參數與證據合成

Scala 3 Context Parameters 允許呼叫點省略某些參數，由編譯器從上下文中的 Given Instances 合成。[R7]

用途包括：

- Type class evidence；
- Ordering；
- Configuration；
- Execution context；
- Capability；
- Dependency injection；
- Type relation proof。

這不是普通型別推導，因為系統必須搜尋：

> Scope
> →
> CandidateTerms

風險包括：

- Action at a distance；
- 匯入改變行為；
- 多個候選歧義；
- 不明顯的執行成本；
- 能力來源不透明；
- Library author 難以預測呼叫端環境。

  Scala 3 將 contextual abstraction 重新設計為 given／using，並限制部分 Scala 2 implicit 行為，官方說明中的目標包括更容易、更安全，並減少令人意外的構造。[R7]

SECTION

十一、隱式轉換與預設化

隱式轉換可能：

- 自動數值提升；
- String 轉 Token；
- Deref／coercion；
- User-defined conversion；
- Truthiness；
- Default numeric type；
- Default type class。

  它能減少樣板，卻也可能：

- 插入成本；
- 改變重載；

- 造成資訊損失；
- 讓錯誤在轉換後才出現；
- 形成模糊 API。

  Scala 3 要求使用 scala.Conversion 表達使用者定義的隱式轉換，並限制 Scala 2 中某些令人意外的轉換來源。[R7]

  預設化則應被視為：

> 約束不足時，語言明確規定的後備選擇。

  它必須文件化，而不是由實作任意決定。

SECTION

十二、公開契約推導

最危險的推導之一，是由函式實作自動決定公開 API 的型別。

例如：

```
public fun result() = implementationExpression
```

若推導型別包含：

- 私有具體類別；
- 過度具體泛型；
- 平台型別；
- 實作細節；
- 複雜結構型別；

  日後修改實作就可能意外改變 ABI、API 或使用者型別檢查。

  Kotlin 的 Explicit API mode 正是要求 Library 的 public／protected declaration 明確標示可見性與型別，以防推導結果意外成為公共契約。[R6]

  因此：

> LocalInference
> not⇒
> PublicContractInference

SECTION

十三、已知事實 K

包括：

- Literal；
- Initializer；
- Parameter；
- Expected type；
- Generic bounds；
- Trait／Class constraints；
- Imported context；
- Control-flow refinement；
- Previous use。

推導系統必須說明它使用哪些事實。

SECTION

十四、約束 C

常見約束：

- Equality；
- Subtyping；

- Assignability；
- Trait satisfaction；
- Lifetime outlives；
- Effect inclusion；
- Ownership uniqueness；
- Overload applicability。

若約束無法被使用者理解，推導錯誤也難以理解。

SECTION

十五、搜尋範圍 S

定義：

> S
> =
> (
> Expression,
> Statement,
> Function,
> Module,
> Imports,
> Package,
> Program,
> Environment
>)

搜尋範圍越大：

- 便利可能上升；
- 非局部依賴增加；

- 編譯成本增加；
- 重構穩定性下降；
- 診斷負擔增加。

SECTION

十六、唯一性 U

理想推導：

> |Solutions(K,C)|=1

若有多個合法解，語言可選擇：

- 報歧義；
- 要求標註；
- 使用明確優先規則；
- 使用預設型別；
- 選擇最特定候選。

優先規則與預設都必須可見、穩定且可預測。

SECTION

十七、歧義政策 A

安全導向機制通常應：

```
ambiguous
→ reject
→ request explicit information
```

而不是：

```
ambiguous
→ silently choose convenient candidate
```

特別是涉及：

- 權限；
- I/O；
- 序列化；
- 安全敏感轉換；
- 網路；
- 金錢；
- 不可逆效果。

SECTION

十八、可解釋性 X

好的推導工具應能回答：

- 推導出什麼？
- 使用了哪些約束？
- 候選從哪裡來？
- 為何排除其他候選？
- 哪一條標註可消除歧義？
- 插入了什麼轉換？
- 最終公開型別是什麼？

可解釋性不能只依 IDE hover；Compiler diagnostics 也應保留最低能力。

SECTION

十九、重構穩定性 R

若局部無語義變更的重構造成：

- 不同重載；
- 不同 Given；
- 不同型別寬化；
- 不同公開型別；
- 不同 Runtime 轉換；

  則推導機制具有低穩定性。

  定義啟發式：

> R
> =
> 1-
> P(
> InferenceChanges
> mid
> SemanticsPreservingRefactor
>)

這不是可直接精確量測的普遍機率，而是測試與比較指標。

SECTION

二十、邊界暴露 B

推導結果若影響：

- 公開 API；
- ABI；
- 序列化；
- 資料庫 Schema；

- FFI；
- Capability；
- 交易效果；

  則 B 高，應提高顯式要求。

SECTION

二十一、局部實作可偏向推導

適合推導：

- 右側明確的區域變數；
- Lambda 參數；
- 泛型呼叫型別引數；
- 明確且唯一的生命週期；
- 無外部可見性的中間型別；
- 可由工具完整顯示的結果。

SECTION

二十二、模組邊界應提高顯式度

適合明示：

- Exported function；
- Public property；
- Interface；
- Trait；
- Module signature；

- FFI；
- Callback contract；
- Error type；
- Capability；
- Ownership transfer；
- Async／effect boundary。

原因是：

> 邊界不是只服務編譯器，也服務人類、文件、版本與其他實作。

SECTION

二十三、不可逆操作應明示

即使能從上下文推導，也不應輕易隱藏：

- 刪除；
- 付款；
- 網路發布；
- 權限提升；
- 磁碟寫入；
- 外部程序；
- 交易提交；
- 安全敏感轉換。

推導型別不等於推導意圖。

SECTION

二十四、依賴與能力應可追溯

Context parameter 可以省略呼叫點重複傳遞，但應能追蹤：

- 具體 Given；
- Scope；
- Provider；
- Lifetime；
- Permission；
- Runtime cost。

能力若只能從不可見上下文取得，會使安全審查困難。

SECTION

二十五、最一般型別推導型

代表：

- ML；
- Hindley–Milner；
- Haskell 的部分核心。

風格：

- 盡可能從程式結構推導 principal type；
- 標註主要用於文件、約束與模組邊界；
- 推導有形式化核心。

優勢：

- 高表達密度；
- 多型抽象自然；
- 減少重複型別。

風險：

- 加入 Subtyping、Overloading、Effects 或高階型別後，principal type 性質可能變得困難或不再成立；
- 錯誤可能在大約束集合後才顯現；
- 型別簽章仍對人類與模組化重要。

SECTION

二十六、區域推導—邊界明示型

代表：

- Rust；
- Kotlin Explicit API；
- 多數現代靜態語言的公共介面政策。

風格：

```
implementation details → infer
public contract → explicit
```

優勢：

- 局部簡潔；
- API 穩定；
- Code review 可見；
- 實作不易意外洩漏。

風險：

- 內外規則不一致；

- 使用者需理解何時必須標註；
- 複雜內部型別仍可能難讀。

SECTION

二十七、約束關係推導型

代表：

- Go 泛型型別引數推導。

風格：

- 只依明確參數與限制關係推導；
- 不足時要求型別引數；
- 優先保留呼叫點簡潔。

優勢：

- 規則範圍較小；
- 與顯式泛型契約配合；
- 失敗容易以補充型別引數修正。

風險：

- 複雜限制可能讓錯誤不直觀；
- API 參數設計會強烈影響可推導性；
- 使用者可能為迎合推導而扭曲介面。

SECTION

二十八、上下文雙向型

代表：

- TypeScript；
- Kotlin Builder inference。

風格：

- 由表達式與期望型別雙向傳遞資訊；
- 讓 Lambda、Object literal 與 DSL 更精簡。

優勢：

- Callback 體驗佳；
- 與既有 JavaScript／JVM API 整合；
- 大量區域標註可省略。

風險：

- 表達式脫離上下文即改變型別；
- 結構型別可能產生複雜結果；
- 推導順序與寬化規則影響結果；
- API 調整可遠端影響 Callback。

SECTION

二十九、上下文證據合成型

代表：

- Scala 3 given／using；
- Type class dictionary synthesis。

風格：

- 將重複傳遞的證據與環境放入型別導向搜尋；
- 讓能力與抽象可組合。

優勢：

- Type class；
- 依賴注入；
- Capability；
- Context propagation；
- 證明物件。

風險：

- 搜尋範圍；
- 歧義；
- 非局部行為；
- 編譯時間；
- 新匯入改變選擇；
- 隱式轉換濫用。

SECTION

三十、顯式控制優先型

代表風格：

- 要求型別、依賴或轉換清楚可見；
- 推導只用於明顯局部重複；
- 歧義立即拒絕；
- 隱式轉換極少。

優勢：

- Code review 容易；
- 行為穩定；
- 編譯器規則較可預測；
- 不依賴完整 IDE。

風險：

- 大量樣板；
- 人工同步錯誤；
- 複雜泛型型別污染程式；
- 使用者重複機器可知資訊。

SECTION

三十一、Damas–Milner：推導的合法性不是「省字」

Damas 與 Milner 的 principal type-scheme 結果指出，在其 ML 型別系統範圍內，推導出的 principal type 能代表其他合法型別方案的最一般形式。[R2]

它的重要性不是少寫幾個型別，而是：

- 推導具有明確語義；
- 結果不是任意；
- Polymorphism 可以在靜態型別下保持高可用性。

但將此成果直接推廣到所有具有：

- Subtyping；
- Ad-hoc overloading；
- Implicit conversion；
- Dependent type；

- Effect；

  的語言，是不正確的。

SECTION

三十二、Haskell：可推導仍保留簽章文化

Haskell 2010 規定，未提供型別簽章的繫結可取得推導出的 principal type；Kind 甚至完全隱式推導。[R3]

然而大型 Haskell 程式仍普遍需要型別簽章，因為簽章同時提供：

- 文件；
- 模組邊界；
- 意圖限制；
- 錯誤定位；
- 防止實作改變外部型別。

  這顯示：

> TypeInference
> +
> ExplicitSignature

不是互斥設計，而是局部與邊界分工。

SECTION

三十三、Rust：推導細節，但明示安全關係

Rust 可推導區域型別，且常見函式生命週期可依 elision rules 省略。[R5]

但當輸入與輸出參考之間的關係不唯一，或資料結構保存參考時，使用者必須明示生命週期。

這反映：

唯一、安全、固定模式

→ 省略
多種合法關係、形成公共語義
→ 明示
Rust 的推導不是全面追求短，而是讓安全關係在需要時重新浮出表面。

SECTION

三十四、Go：推導呼叫，不推導全部契約

Go 泛型型別推導利用：

- 函式引數；
- 函式參數；
- 型別參數；
- Type constraints；

  建立關係並求解型別引數。[R4]

  但泛型宣告本身仍明確列出：

- 型別參數；
- Constraint；
- 函式介面。

  這是一種：

  definition explicit
  call-site inferred
  的配置。

SECTION

三十五、Kotlin：區域便利與公開穩定分層

Kotlin 可以推導已初始化變數的型別；Builder inference 還能利用 Lambda 內部資訊推導泛型 Builder 型別。[R6]

但 Kotlin API Guidelines 建議 Library 使用 Explicit API mode，要求 Public／Protected API 明示型別與可見性，避免 inferred type 因實作調整而改變公共契約。[R6]

此案例非常直接地支持：

> InferenceBenefit_(local)
> not⇒
> InferenceBenefit_(public)

SECTION

三十六、Scala 3：不是取消上下文，而是重新馴化上下文

Scala 3 沒有放棄 implicits 的核心能力，而是將其拆分為：

- Given Instances；
- Using Clauses；
- Context Bounds；
- Extension methods；
- Conversion；
- Context Functions。[R7]

  官方設計指出，Scala 2 的 implicit 同時承擔太多用途，Scala 3 改以更專門構造表達，並清理搜尋規則與令人意外的轉換。[R7]

  這不是「從隱式變顯式」的單線轉換，而是：

> 保留上下文合成能力，同時提高定義、用途與轉換的語義可見性。

SECTION

三十七、TypeScript：上下文塑造表達式型別

TypeScript 可從變數初值推導型別，也能從函式所在位置推導參數型別；這就是 contextual typing。[R8]

它非常適合 JavaScript 生態，因為大量匿名函式與物件字面值都能由 API 契約取得型別。

但結構型別與 contextual typing 也意味：

- 同一表達式在不同位置可能有不同型別；
- void contextual return 等規則可能與直覺不同；
- 工具顯示對理解很重要；
- Library declaration quality 會直接影響使用者推導品質。

SECTION

三十八、Action at a Distance

遠處的：

- Import；
- Given；
- Overload；
- Extension；
- Expected type；

改變本地程式行為。

修正：

- 限制搜尋範圍；
- 顯示來源；
- 歧義拒絕；
- 支援顯式覆寫。

SECTION

三十九、推導洩漏 API

私人實作推導出過度具體的 Public type，造成：

- API 漂移；
- 二進位不相容；
- 私有類別洩漏；
- 使用者依賴偶然細節。

修正：

- 公開邊界顯式；
- API dump；
- 相容性檢查；
- Explicit API mode。

SECTION

四十、推導錯誤距離過長

Compiler 在最後一個約束失敗處報錯，但真正錯誤可能在更早的：

- Literal；
- Generic choice；
- Import；
- Missing annotation；
- Expected type。

修正：

- Constraint trace；
- 最小衝突集合；
- 候選列表；
- 建議標註點；
- 顯示推導結果。

SECTION

四十一、過度寬化或過度具體化

推導型別可能：

- 寬成一般介面，失去資訊；
- 窄成具體類別，洩漏實作；
- 將 Literal 推成不希望的數值型別；
- 將空集合推成 Bottom／Unknown／Any。

需要：

- 明確寬化規則；
- Target typing；
- 顯式 Ascription；
- IDE 顯示。

SECTION

四十二、隱式轉換鏈

多步轉換可能造成：

- 成本不可見；

- 精度損失；
- 不同重載；
- 無限搜尋；
- 錯誤訊息難懂。

原則：

> ConversionDepth
> ≤
> SmallBound

且安全敏感轉換應要求顯式。

SECTION

四十三、編譯成本與搜尋爆炸

上下文搜尋、重載、泛型、Subtyping 與型別層計算組合後，可能造成高編譯成本。

設計應限制：

- 搜尋深度；
- 候選數；
- 遞迴；
- 回溯；
- 全域 Scope；
- 隱式轉換鏈。

SECTION

四十四、工具依賴

若程式離開 IDE 後無法理解：

- 型別；
- 轉換；
- Given；
- 展開；
- 控制效果；

  則工具已成為必要語言層。

  這不一定錯，但必須納入有效語言與部署成本。

SECTION

四十五、局部性原則

> SearchDistancedownarrow
> ⇒
> Predictabilityuparrow

推導優先使用：

- 當前表達式；
- 當前宣告；
- 直接參數；
- 明確匯入 Context。

  避免不可見全域搜尋。

SECTION

四十六、唯一性原則

若解不唯一：

reject
or request annotation
不要以不透明順序選擇。

SECTION

四十七、邊界原則

公開、跨語言、跨程序、跨版本與不可逆邊界提高顯式度。

SECTION

四十八、可逆性原則

可由工具隨時顯示與補回的省略較安全。

例如：

- Hover type；
- Expand implicit；
- Insert annotation；
- Show desugaring；
- Generate explicit arguments。

SECTION

四十九、穩定性原則

推導結果不應因無關重構、匯入順序或 Library 小改動而靜默改變。

SECTION

五十、能力可見原則

會提供權限、I/O、執行緒、交易或安全能力的上下文值，必須可追溯與可覆寫。

SECTION

五十一、公開契約原則

> PublicContract
> ⇒
> ExplicitIntent

即使 Compiler 能推導，也應要求設計者確認。

SECTION

五十二、診斷責任原則

語言每增加一層推導自由，就必須增加一層解釋能力。

> InferencePoweruparrow
> ⇒
> DiagnosticObligationuparrow

SECTION

五十三、收益

> Benefit(I)
> =
> RepetitionRemoved
> +
> LocalClarity
> +
> GenericUsability
> +
> RefactorConvenience
> +

SECTION

五十四、成本

> Cost(I)
> =
> SearchDistance
> +
> Ambiguity
> +
> Instability
> +
> DiagnosticBurden
> +
> CompileCost
> +
> HiddenEffects
> +
> BoundaryLeak

SECTION

五十五、准入判準

候選推導機制應滿足：

> Benefit(I)-Cost(I)>Threshold

並至少通過：

結果是否通常唯一？
是否局部？
能否顯示？
能否顯式覆寫？
是否影響公開契約？
是否插入 Runtime 操作？

歧義是否安全失敗？
版本更新是否穩定？

SECTION

五十六、顯式性指紋

Local type explicitness
Public API explicitness
Effect explicitness
Resource explicitness
Dependency explicitness
Conversion explicitness
Control-flow explicitness
Governance explicitness

SECTION

五十七、推導指紋

Inference scope
Constraint sources
Expected-type usage
Context search
Defaulting
Implicit conversion
Explanation support
Explicit override

SECTION

五十八、設計師比較

後續比較設計者時應問：

- 他主要省略哪種重複資訊？
- 他拒絕推導什麼？
- 他接受多大的搜尋範圍？
- 歧義時拒絕還是預設？
- 公開介面是否要求標註？

- 是否允許使用者定義隱式轉換？
- 工具能否展示推導過程？
- 推導結果是否穩定？
- 他把診斷成本交給誰？
- 推導是否涉及能力與副作用？

SECTION

五十九、不能只給「顯式／隱式」分數

一位設計者可能：

- 高區域型別推導；
- 高公開 API 顯式；
- 低隱式轉換；
- 中度上下文參數；
- 高效果顯式。

單軸評分會抹去真正風格。

SECTION

六十、輸入

designer
language
version_or_period
feature
specification
compiler_diagnostics
API_guidelines
governance_documents

SECTION

六十一、分析管線

重新網路搜尋
→ 省略／推導／搜尋／轉換分類
→ Constraint source 抽取
→ Search scope 抽取
→ 歧義政策
→ Public boundary 檢查
→ Runtime effect 檢查
→ 診斷與工具檢查
→ Refactoring stability
→ 反例搜尋
→ 第二輪事實校對
→ 風格報告

SECTION

六十二、JSON 雛形

```
{
  "mechanism": "context parameter synthesis",
  "category": "contextual evidence search",
  "known_facts": ["required parameter type", "visible givens"],
  "search_scope": ["local", "imports", "implicit scope"],
  "ambiguity_policy": "compile-time error",
  "runtime_operation_inserted": "argument passing",
  "public_boundary_effect": "medium",
  "explainability": {
    "show_selected_candidate": true,
    "show_rejected_candidates": "tool-dependent"
  },
  "style": {
    "local_explicitness": "medium",
    "contextual_inference": "high",
    "implicit_conversion_tolerance": "restricted"
  }
}
```

SECTION

六十三、SKILL 禁止事項

不得：

- 把所有省略稱為型別推導；
- 把隱式轉換稱為零成本；
- 把 Compiler 能推導寫成使用者不需理解；
- 把 IDE Hover 當成語言唯一文件；
- 忽略公開 API 漂移；
- 把歧義預設寫成唯一解；
- 把 Scala 3 說成取消 implicits；
- 把 Rust lifetime elision 說成推導所有生命週期；
- 把 Kotlin 區域推導推廣為公共 API 推導建議；
- 把 TypeScript contextual type 當成與位置無關的固有型別；
- 把 Damas–Milner 結果推廣到任意現代型別系統。

SECTION

六十四、可理解性依賴使用者

專家可能偏好省略複雜型別，初學者可能需要明示；大型團隊也可能制定與語言不同的標註政策。

SECTION

六十五、顯式資訊也可能過時

人工型別、註解與文件可能與實作不同步。推導能消除部分雙重來源。

SECTION

六十六、推導規則會隨語言演化

Kotlin、TypeScript、Rust、Scala 等語言的推導能力會改變。每篇人物或語言個案都必須重新查核版本，不能把某一年行為當成永久規則。

SECTION

六十七、工具顯示不完全可攜

不同 IDE、Compiler 與 Language server 對推導解釋能力不同。PLDST 應分開：

- 語言保證；
- 編譯器行為；
- 工具體驗。

SECTION

六十八、形式唯一不等於人類唯一

即使型別系統具有唯一 principal type，該結果也可能不是 API 設計者真正想承諾的抽象型別。

因此公共邊界仍需要意圖確認。

顯式與推導不是道德對立：

- 顯式不一定清楚；
- 推導不一定神祕；
- 省略不一定改變語義；
- 隱式轉換卻可能真的插入操作；
- 局部推導可能提高可讀性；
- 公開契約推導可能製造長期不穩定。

本文提出：

> cal-I
> =
> (

並將自動性分成：

> 【表面省略
> +
> 區域推導
> +
> 雙向推導
> +
> 約束求解
> +
> 上下文合成
> +
> 隱式轉換／預設
> +
> 公開契約推導】

成熟的語言設計應遵守：

> 【局部、唯一、穩定、可解釋
> ⇒
> 可優先推導】

以及：

> 【跨邊界、具能力、改控制流、影響安全、形成契約
> ⇒
> 提高顯式性】

因此，PLDST 不再只問某位設計者「偏顯式還是偏隱式」，而要分析：

> 他允許機器從哪裡取得資訊、搜尋多遠、歧義時如何處理、哪些推導只服務局部便利、哪些推導會改變程式操作，以及哪些資訊即使可被推導，仍必須由設計者明確承諾。
> 最終原則為：

> 【讓機器消除重複
> ∧
> 讓人類保留意圖
> ∧
> 不讓推導掩蓋權力、效果與契約】

語言：
版本／時期：
機制：
類別：省略／推導／搜尋／轉換／預設
既知事實：
約束：
搜尋範圍：
唯一性：
歧義政策：
插入 Runtime 操作：
公開邊界影響：
安全／能力影響：
工具可見性：
顯式覆寫方式：
重構穩定性：
編譯成本：
主要收益：
主要代價：
證據：
信心：

[R1] Robin Milner, “A Theory of Type Polymorphism in Programming,” Journal of Computer and System Sciences 17(3), 1978, pp. 348–375.

— ML 型別紀律、Algorithm W 與編譯期多型型別檢查。

[R2] Luís Damas and Robin Milner, “Principal Type-Schemes for Functional Programs,” POPL 1982.

— Principal type-scheme 的形式結果與最一般型別性質。

[R3] Simon Marlow (ed.), Haskell 2010 Language Report, 2010.

— Inferred／principal types、型別簽章、Kind inference、Type classes 與預設邊界。

[R4] Go Project, The Go Programming Language Specification, section “Type inference”; Go generics design and tutorial documents.

— 由 Assignability、型別參數與限制關係推導泛型型別引數。

[R5] Rust Project, The Rust Reference, “Inferred Type” and “Lifetime Elision”; The Rust Programming Language, lifetime chapters.

— 區域型別推導、資訊不足時報錯、生命週期省略規則與明示邊界。

[R6] Kotlin Documentation, “Basic Syntax,” “Using Builders with Builder Type Inference,” and “API Guidelines: Simplicity.”

— 區域型別推導、Builder inference，以及 Explicit API mode 對公共型別與可見性的要求。

[R7] Scala 3 Reference, “Contextual Abstractions,” “Changes in Implicit Resolution,” “Implicit Conversions,” “Using Clauses,” and Migration Guide.

— Given／Using、Context synthesis、Implicit resolution 改革、Conversion，以及非區域 implicit 定義的明示型別要求。

[R8] TypeScript Handbook, “Type Inference,” “Everyday Types,” and “TypeScript for Functional Programmers.”

— Best common type、Contextual typing、由位置與其他引數反向推導。

[E-L] Local explicitness
[E-B] Boundary explicitness
[E-F] Effect explicitness
[E-C] Capability explicitness
[E-X] Conversion explicitness
[I-L] Local inference
[I-B] Bidirectional inference
[I-G] Generic inference
[I-R] Region／lifetime inference
[I-C] Context synthesis
[I-D] Defaulting
[I-P] Public contract inference

SECTION

D.1 Milner 與 Damas–Milner

已重新核對 1978 年 Milner 原始論文與 1982 年 Damas–Milner 論文：

- Milner 提出的是在一個簡化程式語言與特定多型型別紀律中的編譯期型別檢查演算法；

- Damas–Milner 證明的是該體系中的 principal type-scheme 性質；
- 本文未把 principal type 性質推廣到任意具有 Subtyping、使用者定義隱式轉換、效果或依賴型別的現代語言。

SECTION

D.2 Haskell 2010

已核對 Haskell 2010 Report：

- 未提供型別簽章的繫結，可在規定的 declaration group 與 generalization 規則下取得 inferred／principal type；
- 若提供簽章，而定義無法推導出相容型別，則為靜態錯誤；
- 本文保留 Monomorphism Restriction、Type class context 與 Defaulting 等限制，沒有將 Haskell 描述成無條件完整全域推導。

SECTION

D.3 Go 泛型型別推導

已核對 2026 年 Go Specification：

- 泛型函式可以省略部分或全部型別引數；
- 推導依據包括函式引數與參數間的 assignability，以及型別引數是否滿足 Constraint；
- 規格將問題表達為型別方程與 Unification；
- 若無法推得全部必要型別引數，推導失敗且程式無效。

因此本文沒有把 Go 推導描述成啟發式選擇。

SECTION

D.4 Rust 區域推導與 Lifetime Elision

已核對 Rust Reference：

- _ inferred type 只要求編譯器根據周圍資訊推導可能的型別；
- inferred type 不能用於 item signatures；
- lifetime elision 只適用於規格列出的函式、函式指標與 Closure trait 等位置；
- 無法依規則推導時，省略生命週期是編譯錯誤。

  因此本文沒有把 Rust 描述成會自動推導所有公開生命週期與資料結構關係。

SECTION

D.5 Kotlin 區域推導與 Explicit API

已核對 Kotlin 官方文件：

- 已初始化的區域變數通常可以省略型別；
- Builder inference 能從 Lambda receiver 中的呼叫收集資訊；
- Kotlin Library API Guidelines 建議使用 Explicit API mode；
- 該模式要求公共宣告明示可見性，並要求公共函式與屬性定義型別，以避免 inferred type 意外改變 API。

  本文因此將局部便利與公共契約分開，而未反對 Kotlin 的一般型別推導。

SECTION

D.6 Scala 3 Contextual Abstractions

已核對 Scala 3 Reference、Contextual Abstractions、Implicit Resolution 與 Conversion 文件：

- Scala 3 沒有取消 term inference 或 contextual abstraction；
- Given／Using 重新區分原本由 implicit 同時承擔的多種用途；
- 官方設計目標明確包括馴化強大 constructs、降低意外行為與改善安全性；

- Implicit scope 與 resolution 規則被清理；
- 使用者定義的隱式轉換需以 Conversion 等明確機制表達，並受語言功能開關約束；
- 非區域 implicit value 與 implicit method 必須明示型別，區域 block 內有例外。

本文已避免將這些規則錯寫成「Scala 3 的所有 Given 都不能推導」或「Scala 3 已移除 Implicit」。

SECTION

D.7 TypeScript Contextual Typing

已核對 TypeScript 官方 Handbook：

- Contextual typing 是由表達式所在位置提供型別；
- 函式運算式參數可由指派位置或 API 契約推導；
- Best common type 找不到單一超型別時，可能形成 Union；
- Contextual void 等規則顯示，推導型別與表達式固有執行行為不能簡化為完全相同概念。

本文將 TypeScript 分類為上下文雙向風格，而非主張其具有 Damas–Milner 式 principal type。

SECTION

D.8 「猜」的用語邊界

標題中的「替使用者猜多少」是通俗修辭。

本文正式區分：

規則化省略
約束式推導
上下文證據搜尋
明確預設
啟發式或不透明選擇

只有最後一類接近一般語義中的「猜」。形式推導不應因標題措辭而被誤解為任意決定。