Practical Unification for Dependent Type Checking

Víctor López Juan

Supervisor: Nils Anders Danielsson

Department of Computer Science and Engineering
Chalmers University of Technology

Licentiate Seminar
October 2nd, 2020

The type of a program

- Programs calculate outputs from inputs

Example (Type of a program)

The program inputs a list of numbers
and outputs a list of numbers

- Dependent types can specify how the output depends on the input.

Example (Dependent type)

The program inputs a list of numbers and
outputs a list with the numbers in the input, sorted from lowest to highest

- A type checker can verify that a program has a given type
- Dependent types can include calculations
  $\Rightarrow$ Dependent type checking is challenging

Examples of dependent types

Declarations

$$
\begin{aligned}
&\mathrm{Nat} && : \mathrm{Set} \\
&\mathrm{Int} && : \mathrm{Set} \\
&\mathrm{Vec} && : (n : \mathrm{Nat}) \to (A : \mathrm{Set}) \to \mathrm{Set} \\
&\mathit{append} && : \{m : \mathrm{Nat}\} \to \{n : \mathrm{Nat}\} \to \mathrm{Vec}\ m\ \mathrm{Int} \to \\
& && \quad \mathrm{Vec}\ n\ \mathrm{Int} \to \mathrm{Vec}\ (m + n)\ \mathrm{Int} \\
&\mathit{repeat} && : \{n : \mathrm{Nat}\} \to \mathrm{Int} \to \mathrm{Vec}\ n\ \mathrm{Int}
\end{aligned}
$$

Examples

$$\vdash 2 : \mathrm{Nat}$$
$$\vdash 1 + 1 : \mathrm{Nat}$$
$$\vdash [-1, 1] : \mathrm{Vec}\ 2\ \mathrm{Int}$$

$$\vdash \mathit{append}\ [-1, 1]\ [1] : \mathrm{Vec}\ (1 + 2)\ \mathrm{Int}$$
$$\vdash \mathit{repeat}\ 1 : \mathrm{Vec}\ 2\ \mathrm{Int}$$

- Terms and types share the same syntax
- Arguments surrounded by $\{\ \ldots\ \}$ are implicit.
- For simplicity, $\Sigma; \Gamma \vdash \mathrm{Set} : \mathrm{Set}$.

First design decision: Uniqueness

Implicit arguments introduce multiple ways of typechecking a term:

Example (Typechecking with implicit arguments)

$$
\begin{aligned}
&\mathit{append} : \{m : \mathrm{Nat}\} \to \{n : \mathrm{Nat}\} \to \mathrm{Vec}\ m\ \mathrm{Int} \to \\
&\qquad \mathrm{Vec}\ n\ \mathrm{Int} \to \mathrm{Vec}\ (m + n)\ \mathrm{Int} \\
&\mathit{repeat}\ : \{n : \mathrm{Nat}\} \to \mathrm{Int} \to \mathrm{Vec}\ n\ \mathrm{Int} \\
&\vdash^{?} \mathit{append}\ (\mathit{repeat}\ 1)\ (\mathit{repeat}\ (-1)) : \mathrm{Vec}\ (2 + 1)\ \mathrm{Int}
\end{aligned}
$$

Solutions

1. $\mathit{append}\ \{2\}\ \{1\}\ (\mathit{repeat}\ \{2\}\ 1)\ (\mathit{repeat}\ \{1\}\ (-1)) : \mathrm{Vec}\ (2 + 1)\ \mathrm{Int}$
   Program result: $[1, 1, -1]$
2. $\mathit{append}\ \{0\}\ \{3\}\ (\mathit{repeat}\ \{0\}\ 1)\ (\mathit{repeat}\ \{3\}\ (-1)) : \mathrm{Vec}\ (2 + 1)\ \mathrm{Int}$
   Program result: $[-1, -1, -1]$

Question

Allow non-unique (and choose (1.)), or reject problem as ambiguous?

Uniqueness on implementations of dependent type checkers

Key: ✅: Fulfilled –: Unfulfilled partly or by design ❌: Unfulfilled

	Coq	Agda		Ours
1. Uniqueness	–	✅		✅
2. Reordering	–	✅		
3. Well-typedness	✅	❌		

- Not enforce uniqueness (f.ex. Coq): Less frustration.
- Enforce uniqueness (f.ex. Agda): More predictable.
- We choose to enforce uniqueness of our solutions (1.).
- Uniqueness allows more flexibility (2.), but this flexibility introduces challenges (3.)
- We wish to reconcile both.

Problem: Dependent type-checking with implicits

A possible type-checking problem.
(We use $U \to V$, $(x : U) \to V$ or $\{x : U\} \to V$ for $\Pi U V$).

$$\begin{array}{l} \mathbb{A} : \mathrm{Set}, \\ \mathbb{id} : \{X : \mathrm{Set}\} \to X \to X; \\ \cdot \vdash^{?} \lambda x.(\mathbb{id}\ x) : \mathbb{A} \to \mathbb{A} \end{array}$$

Goal: Find Σ^{sol} such that:

- Σ^{sol} instantiates all meta-variables in Σ
- $\Sigma^{\mathsf{sol}}; \cdot \vdash \lambda x.\mathbb{id}\ \{\alpha\ x\}\ x : \mathbb{A} \to \mathbb{A}$

Solution: $\Sigma^{\mathsf{sol}} \stackrel{\mathsf{def}}{=} \mathbb{A} : \mathrm{Set}, \mathbb{id} : X : \mathrm{Set} \to X \to X, \alpha := \lambda.\mathbb{A} : \mathbb{A} \to \mathrm{Set}.$

- How to find a unique Σ^{sol} in general?

Problem: Dependent type-checking with implicits

First, we replace each implicit argument with a metavariable:

$$\begin{aligned}\Sigma \stackrel{\text{def}}{=} \mathbb{A} &: \mathrm{Set}\\ \mathbb{id} &: \{X : \mathrm{Set}\} \to X \to X\\ \alpha &: \mathbb{A} \to \mathrm{Set}\\ \Sigma; \cdot \vdash^{?} \lambda x.(\mathbb{id}\ \{\alpha\ x\}\ x) &: \mathbb{A} \to \mathbb{A}\end{aligned}$$

Goal: Find Σ^{sol} such that:

- Σ^{sol} instantiates all meta-variables in Σ
- $\Sigma^{\mathsf{sol}}; \cdot \vdash \lambda x.\mathbb{id}\ \{\alpha\ x\}\ x : \mathbb{A} \to \mathbb{A}$

Solution: $\Sigma^{\mathsf{sol}} \stackrel{\text{def}}{=} \mathbb{A} : \mathrm{Set}, \mathbb{id} : X : \mathrm{Set} \to X \to X, \alpha := \lambda.\mathbb{A} : \mathbb{A} \to \mathrm{Set}.$

- How to find a unique Σ^{sol} in general?

Type checking as unification

- Dependent type checking ($\Sigma; \Gamma \vdash^{?} t : A$) with implicit arguments is a higher-order unification problem.

Example (Higher-order unification problem)

$$\Sigma \stackrel{\text{def}}{=} \mathbb{A} : \mathrm{Set}, \mathbb{id} : \{X : \mathrm{Set}\} \to X \to X, \alpha : \mathbb{A} \to \mathbb{A}$$
$$\Sigma'; x : \mathbb{A} \vdash \alpha\, x : \mathrm{Set} \approx \mathbb{A} : \mathrm{Set} \quad (\Sigma' \text{ extends } \Sigma)$$

Goal: Find Σ^{sol} such that:

- Σ^{sol} is an extension of Σ' and instantiates all meta-variables
- $\Sigma^{\mathsf{sol}}; \cdot \vdash \mathrm{Set} \equiv \mathrm{Set}$ **type** and $\Sigma^{\mathsf{sol}}; x : \mathbb{A} \vdash \alpha\, x \equiv \mathbb{A} : \mathrm{Set}$

- Whether Σ^{sol} exists is undecidable in general.

Definition (Pattern fragment, Miller 1991)

If $\Sigma^{\mathsf{sol}}; \Gamma \vdash \alpha\, \vec{x} \equiv t : A$ ($\vec{x}$ distinct) then there is a unique solution for α.

Solution: $\Sigma^{\mathsf{sol}} \stackrel{\text{def}}{=} \mathbb{A} : \mathrm{Set}, \mathbb{id} : X : \mathrm{Set} \to X \to X, \alpha := \lambda y.\mathbb{A} : \mathbb{A} \to \mathbb{A}$.

Pattern unification

- Dependent type checking ($\Sigma; \Gamma \vdash^? t : A$) with implicit arguments is a higher-order unification problem.

Example (Higher-order unification problem)

$$\Sigma \stackrel{\text{def}}{=} \mathbb{A} : \mathrm{Set}, \mathbb{id} : \{X : \mathrm{Set}\} \to X \to X, \alpha : \mathbb{A} \to \mathbb{A}$$
$$\Sigma'; x : \mathbb{A} \vdash \alpha\, x : \mathrm{Set} \approx \mathbb{A} : \mathrm{Set} \quad (\Sigma' \text{ extends } \Sigma)$$

Goal: Find Σ^{sol} such that:

- Σ^{sol} is an extension of Σ' and instantiates all meta-variables
- $\Sigma^{\mathsf{sol}}; \cdot \vdash \mathrm{Set} \equiv \mathrm{Set}$ **type** and $\Sigma^{\mathsf{sol}}; x : \mathbb{A} \vdash \alpha\, x \equiv \mathbb{A} : \mathrm{Set}$

- Whether Σ^{sol} exists is undecidable in general.

Definition (Pattern fragment, Miller 1991)

If $\Sigma^{\mathsf{sol}}; \Gamma \vdash \alpha\, \vec{x} \equiv t : A$ ($\vec{x}$ distinct) then there is a unique solution for α.

Solution: $\Sigma^{\mathsf{sol}} \stackrel{\text{def}}{=} \mathbb{A} : \mathrm{Set}, \mathbb{id} : X : \mathrm{Set} \to X \to X, \alpha := \lambda y.\mathbb{A} : \mathbb{A} \to \mathbb{A}$.

2. Reordering constraints vs. 3. Well-typedness

$$\Sigma \stackrel{\text{def}}{=} \alpha : \mathrm{Nat} \to \mathrm{Set}$$

$$\Sigma; \cdot \vdash \lambda y.\alpha\, y \;\; : \mathrm{Nat} \to \mathrm{Set} \approx$$
$$\lambda y.\mathrm{Nat} : \alpha\, 42 \to \mathrm{Set}$$

$$\rightsquigarrow$$

$$\Sigma; \cdot \vdash \mathrm{Nat} \approx \alpha\, 42 : \mathrm{Set} \qquad ①$$

$$\Sigma; y : ? \vdash \alpha\, y \approx \mathrm{Nat} : \mathrm{Set} \qquad ②$$

- ① does not have a unique solution:
  $\alpha := \lambda y.(\mathrm{if}\ (y < 100)\ \mathrm{then}\ \mathrm{Nat}\ \mathrm{else}\ \mathrm{Bool}) : \mathrm{Nat} \to \mathrm{Set}$
  $\alpha := \lambda y.(\mathrm{if}\ (y > 30)\ \mathrm{then}\ \mathrm{Nat}\ \mathrm{else}\ \mathrm{Bool}) : \mathrm{Nat} \to \mathrm{Set}$
  ...
- What should $?$ be in ②?

2. Reordering constraints vs. 3. Well-typedness

$$\Sigma \stackrel{\text{def}}{=} \alpha : \mathrm{Nat} \to \mathrm{Set}$$

$$\Sigma; \cdot \vdash \mathrm{Nat} \approx \alpha\, 42 : \mathrm{Set} \qquad ①$$

$$\Sigma; y : ? \vdash \alpha\, y \approx \mathrm{Nat} : \mathrm{Set} \qquad ②$$

- ① does not have a unique solution:
  $\alpha := \lambda y.(\mathrm{if}\ (y < 100)\ \mathrm{then}\ \mathrm{Nat}\ \mathrm{else}\ \mathrm{Bool}) : \mathrm{Nat} \to \mathrm{Set}$
  $\alpha := \lambda y.(\mathrm{if}\ (y > 30)\ \mathrm{then}\ \mathrm{Nat}\ \mathrm{else}\ \mathrm{Bool}) : \mathrm{Nat} \to \mathrm{Set}$
  ...
- What should $?$ be in ②?

Results (on a more complex variant of this example)

Coq: Stuck	Agda: Solve ② (ignore $?$)
$\mathrm{Nat} \neq \alpha\, 42$	$[\alpha := \lambda y.\mathrm{Nat} : \mathrm{Nat} \to \mathrm{Set}]$

- Ignoring types can result in bugs.

Twin variables

- Variables usually have one type: $\Sigma; \Gamma, x : A, \Delta \vdash x : A$
- Gundry and McBride (2013) propose giving each variable two types:

$$\Sigma; \Gamma, \hat{x} : A \ddagger B, \Delta \vdash \acute{x} : A \qquad \Sigma; \Gamma, \hat{x} : A \ddagger B, \Delta \vdash \grave{x} : B$$

Example

$$\Sigma := \alpha : \mathrm{Nat} \to \mathrm{Set}$$
$$\Sigma; \cdot \vdash \mathrm{Nat} \approx \alpha\, 42 : \mathrm{Set}$$
$$\Sigma; \hat{y} : \mathrm{Nat} \ddagger (\alpha\, 42) \vdash \alpha\, \acute{y} \approx \mathrm{Nat} : \mathrm{Set}$$

	Coq	Agda	Twin variables	Ours
1. Uniqueness	–	✓	✓	✓
2. Reordering	–	✓	✓	✓
3. Well-typedness	✓	✗	✓	✓
4. Minimal changes	✓	✓	–	
5. Evaluation	✓	✓	–	

Goal #4: Minimal changes (1/2)

- We observe that type variable annotations ($\acute{x}$, $\grave{x}$) are not critical.

Definition (Well-typed constraint)

$\Sigma; \Gamma \ddagger \Gamma' \vdash t \approx u : A \ddagger B$ is well-typed iff $\Sigma; \Gamma \vdash t : A$ and $\Sigma; \Gamma' \vdash u : B$

Example

$$\Sigma := \alpha : \mathrm{Nat} \to \mathrm{Set}$$
$$\Sigma; \cdot \vdash \mathrm{Nat} \approx \alpha\, 42 : \mathrm{Set} \ddagger \mathrm{Set}$$
$$\Sigma; y : \mathrm{Nat} \ddagger (\alpha\, 42) \vdash \alpha\, y \approx \mathrm{Nat} : \mathrm{Set} \ddagger \mathrm{Set}$$

We support many common rules. For example:

Rule (λ-type)

$\Sigma; \Gamma \ddagger \Gamma' \vdash \lambda.t \approx \lambda.u : \Pi A B \ddagger \Pi A' B'$
$\rightsquigarrow\ \Sigma; \Gamma \ddagger \Gamma', A \ddagger A' \vdash t \approx u : B \ddagger B'$

Goal #4: Minimal changes (2/2)

Rule (Syntactic equality, homogeneous version)

$\Sigma; \Gamma \vdash t \approx t : A \rightsquigarrow \Sigma; \square$

- This is sound: For any $\Sigma; \Gamma \vdash t : A$ we have $\Sigma; \Gamma \vdash t \equiv t : A$.

Rule (Syntactic equality, heterogeneous version)

$\Sigma; \Gamma \ddagger \Gamma' \vdash t \approx t : A \ddagger A' \rightsquigarrow \Sigma; \square$

Definition (Heterogeneous equality)

We have $\Sigma; \Gamma_1 \ddagger \Gamma_2 \vdash t \equiv\{v\}\equiv u : A \ddagger B$ if and only if:

i) $\Sigma; \Gamma_1 \vdash t \equiv v : A$,

ii) $\Sigma; \Gamma_2 \vdash u \equiv v : B$,

iii) and $\mathrm{FV}(v) \subseteq \mathrm{FV}(t) \cap \mathrm{FV}(u)$

- For $\Sigma; \Gamma \vdash t : A$ and $\Sigma; \Gamma' \vdash t : A'$ we have $\Sigma; \Gamma \ddagger \Gamma' \vdash t \equiv\{t\}\equiv t : A \ddagger A'$

Goal #5: Evaluation

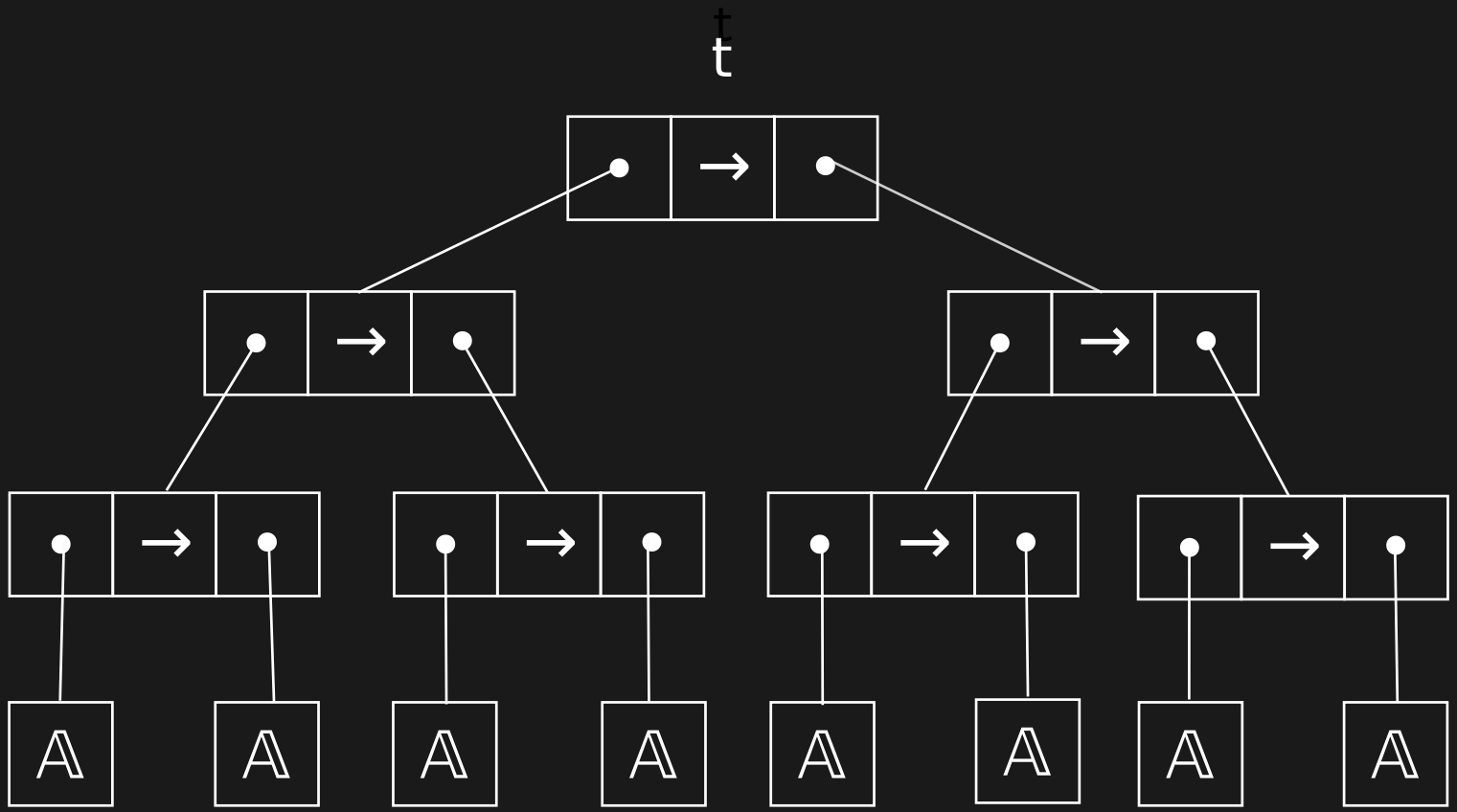
	Coq	Agda	Twin variables	Ours
1. Uniqueness	–	✓	✓	✓
2. Reordering	–	✓	✓	✓
3. Well-typedness	✓	✗	✓	✓
4. Minimal changes	✓	✓	–	✓
5. Evaluation	✓	✓	–	

- We implement the approach into Tog (Mazzoli et al.).
- The internal syntax of terms is respected.
- For acceptable performance, we improve the following:
  - Elaboration
  - Term representation: **Hash consing**
  - Blockers for constraint scheduling

Hash consing

$$\mathrm{id} : \{X : \mathrm{Set}\} \to X \to X$$
$$\mathrm{id}\ \{\alpha\}\ \mathrm{id}\ \mathrm{id}\ :\ \mathbb{A} \to \mathbb{A}$$

$$\alpha := ((\mathbb{A} \to \mathbb{A}) \to (\mathbb{A} \to \mathbb{A})) \to ((\mathbb{A} \to \mathbb{A}) \to (\mathbb{A} \to \mathbb{A})) : \mathrm{Set}$$



Hash consing

$$\begin{aligned} &\mathrm{id} : \{X : \mathrm{Set}\} \to X \to X \\ &\mathrm{id}\ \{\alpha\}\ \mathrm{id}\ \mathrm{id}\ \ :\ \ \mathbb{A} \to \mathbb{A} \end{aligned}$$

$$\alpha := ((\mathbb{A} \to \mathbb{A}) \to (\mathbb{A} \to \mathbb{A})) \to ((\mathbb{A} \to \mathbb{A}) \to (\mathbb{A} \to \mathbb{A})) : \mathrm{Set}$$

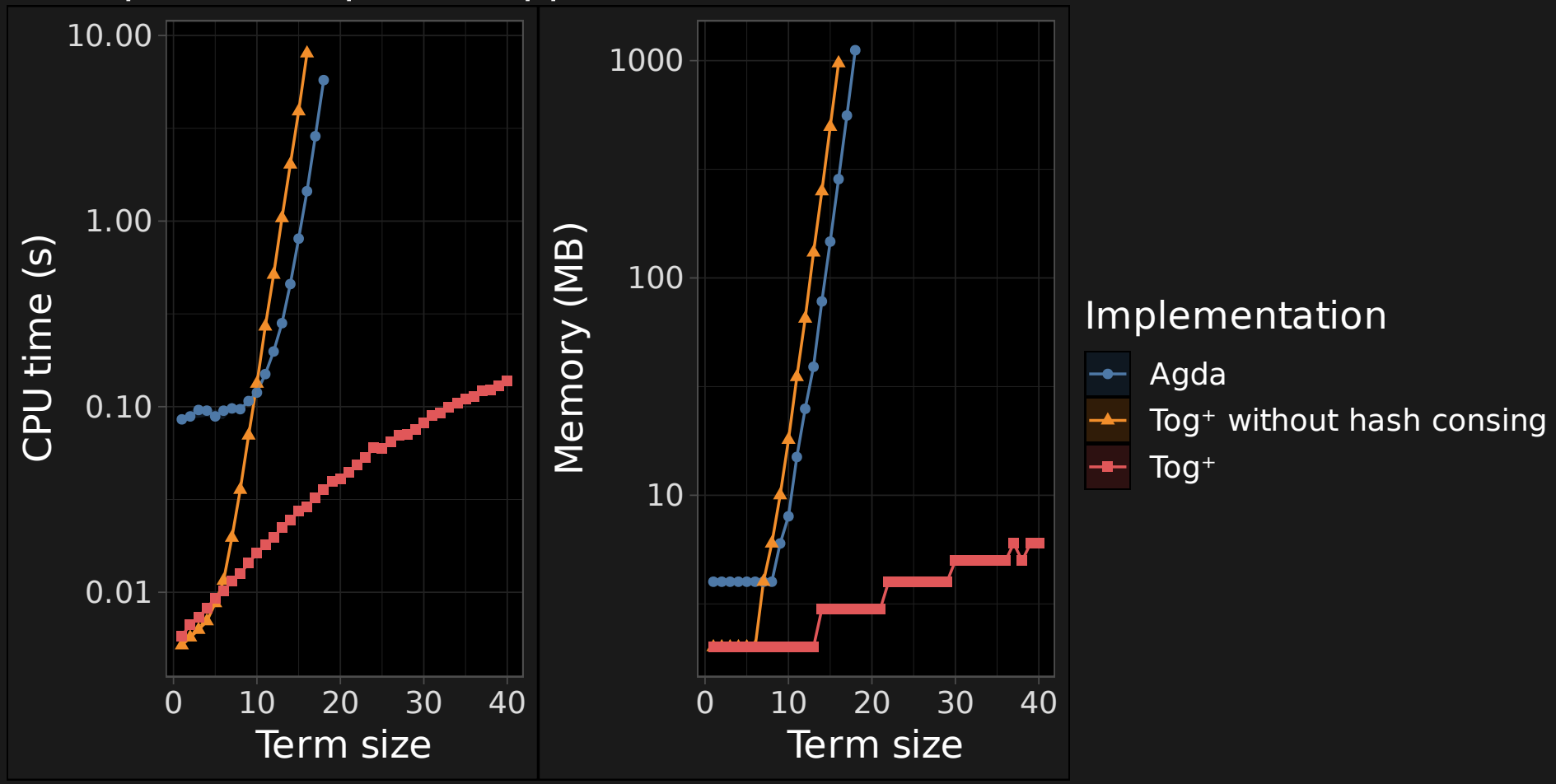
- Each unique term is given a numeric identifier:

Identifier	Term
t : $\#0$	$\#1 \to \#1$
$\#1$	$\#2 \to \#2$
$\#2$	$\#3 \to \#3$
$\#3$	$\mathbb{A}$

- Reduces memory usage.
- Accelerates equality checks, caching.

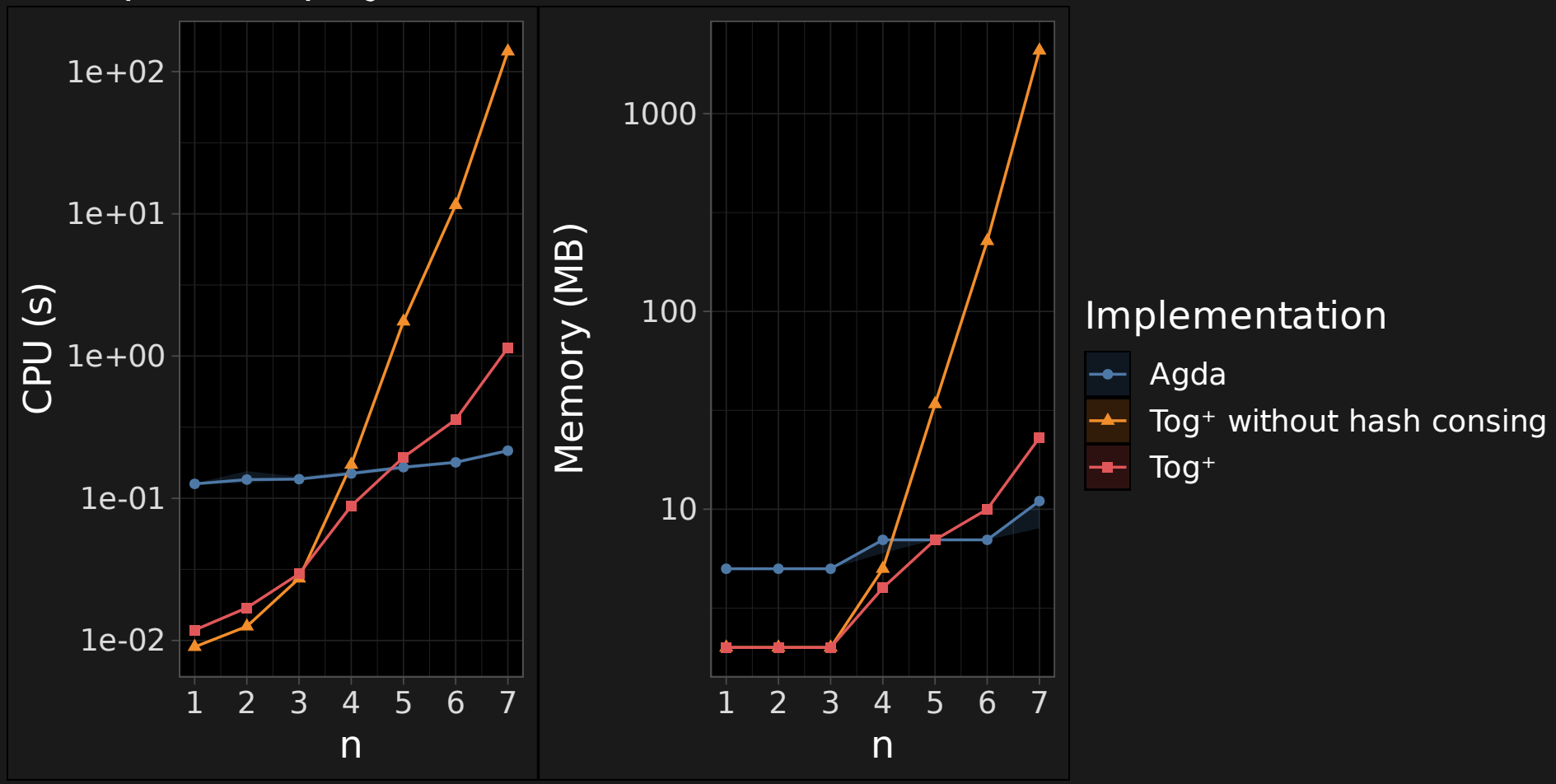
Hash consing: Evaluation

Hash-consing gives a straightforward performance boost.
Example with repeated application of id:



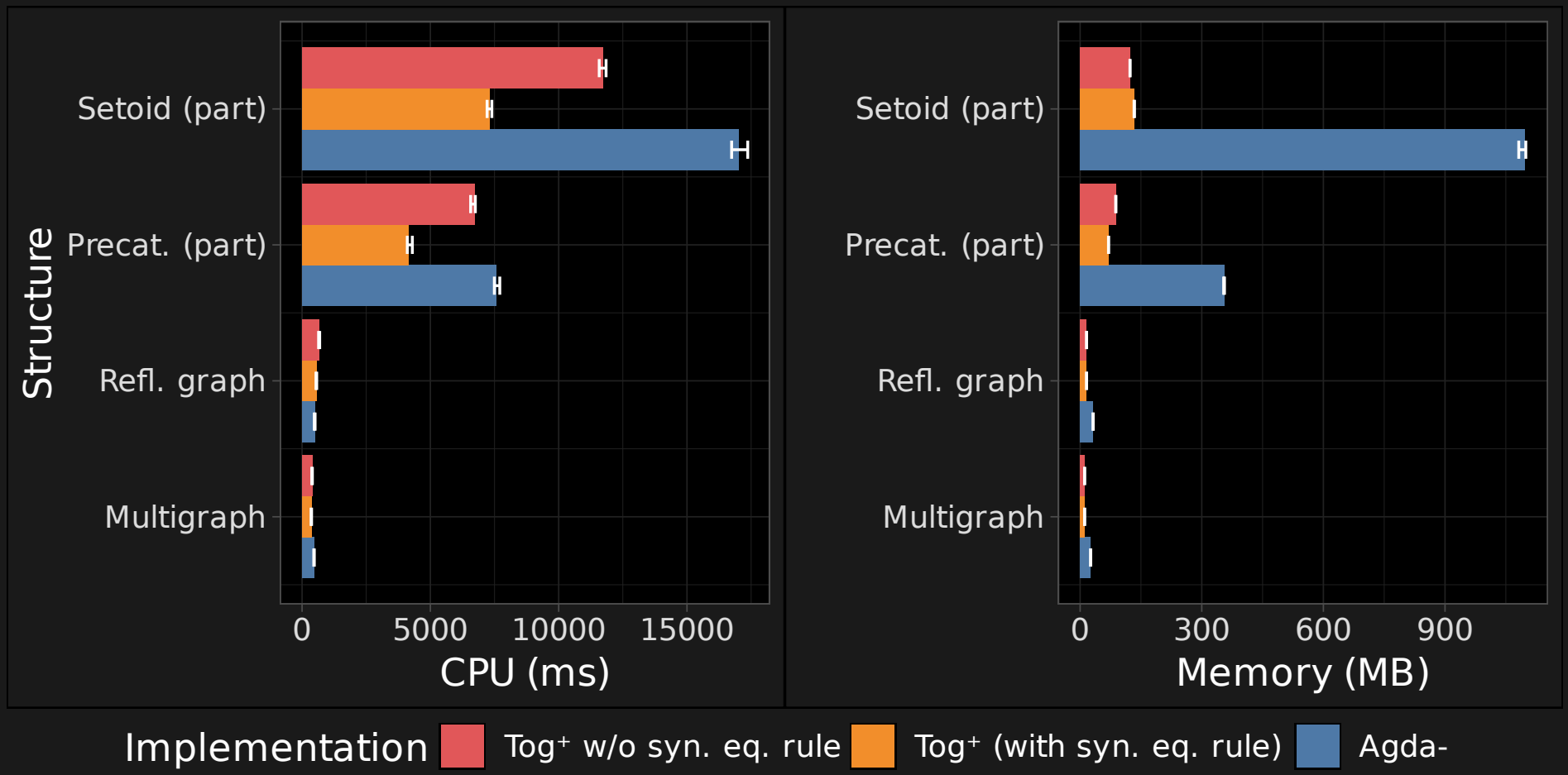
Hash consing: Evaluation

In other cases special case optimizations are better.
Example with *projection-like functions*:



Evaluation: Case study

- Embedded type theory inspired by McBride
- Many constraints, term-before-type unification.



Conclusions

	Coq	Agda	Twin variables	Ours
1. Uniqueness	–	✅	✅	✅
2. Reordering	–	✅	✅	✅
3. Well-typedness	✅	❌	✅	✅
4. Minimal changes	✅	✅	–	✅
5. Evaluation	✅	✅	–	✅

- We demonstrate that a simplified version of twin types in practice.
- We introduce a heterogeneous notion of equality, and a syntactic equality rule which can sometimes improve performance.
- Future work:
  1. Implement in Agda
  2. Test more examples

References I

- L. Peter Deutsch. *An interactive program verifier*. Xerox, Palo Alto Research Center, 1973. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.696.5498.
- Adam Gundry. "Type Inference, Haskell and Dependent Types". PhD thesis. Department of Computer and Information Sciences, University of Strathclyde, 2013. URL: http://adam.gundry.co.uk/pub/thesis/.
- Francesco Mazzoli et al. *Tog - A prototypical implementation of dependent types*. 2017. URL: https://github.com/bitonic/tog.
- Conor McBride. "Outrageous but Meaningful Coincidences: Dependent type-safe syntax and evaluation". In: *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*. WGP'10. 2010. DOI: 10.1145/1863495.1863497.

References II

- Dale Miller. “A logic programming language with lambda-abstraction, function variables, and simple unification”. In: *Journal of logic and computation* 1.4 (1991), pp. 497–536. DOI: 10.1093/logcom/1.4.497.
- Jason Reed. “Higher-Order Constraint Simplification in Dependent Type Theory”. In: *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*. LFMTP ’09. 2009, pp. 49–56. DOI: 10.1145/1577824.1577832.
- Zhong Shao, Christopher League, and Stefan Monnier. “Implementing Typed Intermediate Languages”. In: *ICFP ’98*. 1998. DOI: 10.1145/289423.289460.

Our setting

$$
\begin{array}{lll}
t, u, v, A, B, T, U, \ldots ::= & & \\
\quad \mathrm{Set} & & \text{universe} \\
\mid \mathrm{Bool} & & \text{boolean type} \\
\mid \Pi A B & & \text{function type} \\
\mid \Sigma A B & & \text{product type} \\
\mid \mathrm{true} \mid \mathrm{false} & & \text{booleans} \\
\mid \lambda.t & & \text{abstraction} \\
\mid \langle t, u \rangle & & \text{pair} \\
\mid h\,\vec{e} & & \text{neutral term}
\end{array}
$$

$$
\begin{array}{lll}
h ::= x, y, z, \ldots & & \text{variable} \\
\quad \mid \alpha, \beta, \ldots & & \text{metavariable} \\
\quad \mid \mathbb{a}, \mathbb{b}, \ldots & & \text{atom} \\
\quad \mid \mathrm{if} & & \text{boolean recursor} \\
x ::= 0, 1, 2, \ldots & & \text{deBruijn indices} \\
e ::= & & \textit{eliminator} \\
\quad \mid t & & \text{term application} \\
\quad \mid .\pi_1 \mid .\pi_2 & & \text{projections}
\end{array}
$$

- Hereditary substitution: $t[u] \Downarrow v$. For example, $0[u] \Downarrow u$
- Signatures: $\Sigma ::= \cdot \mid \Sigma, \alpha : A \mid \Sigma, \alpha := t : A \mid \Sigma, \mathbb{a} : A$ ($\mathrm{FV}(A) = \mathrm{FV}(t) = \emptyset$) and Contexts: $\Gamma ::= \cdot \mid \Gamma, A$
- Judgments: Σ **sig**, $\Sigma \vdash \Gamma$ **ctx**, $\Sigma; \Gamma \vdash A$ **type**, $\Sigma; \Gamma \vdash t : A$, $\Sigma; \Gamma \vdash A \equiv B$ **type**, $\Sigma; \Gamma \vdash t \equiv u : A$.
- For simplicity, $\Sigma; \Gamma \vdash \mathrm{Set} : \mathrm{Set}$.

Our setting

$$
\begin{array}{lll}
t, u, v, A, B, T, U, \ldots ::= & & \\
\quad \mathrm{Set} & & \text{universe} \\
\mid \mathrm{Bool} & & \text{boolean type} \\
\mid \Pi A B & & \text{function type} \\
\mid \Sigma A B & & \text{product type} \\
\mid \mathrm{true} \mid \mathrm{false} & & \text{booleans} \\
\mid \lambda.t & & \text{abstraction} \\
\mid \langle t, u \rangle & & \text{pair} \\
\mid h\,\vec{e} & & \text{neutral term}
\end{array}
$$

$$
\begin{array}{lll}
h ::= x, y, z, \ldots & & \text{variable} \\
\quad \mid \alpha, \beta, \ldots & & \text{metavariable} \\
\quad \mid \mathbb{a}, \mathbb{b}, \ldots & & \text{atom} \\
\quad \mid \mathrm{if} & & \text{boolean recursor} \\
x ::= 0, 1, 2, \ldots & & \text{deBruijn indices} \\
e ::= & & \textit{eliminator} \\
\quad \mid t & & \text{term application} \\
\quad \mid .\pi_1 \mid .\pi_2 & & \text{projections}
\end{array}
$$

- Hereditary substitution: $t[u] \Downarrow v$. For example, $0[u] \Downarrow u$
- Signatures: $\Sigma ::= \cdot \mid \Sigma, \alpha : A \mid \Sigma, \alpha := t : A \mid \Sigma, \mathbb{a} : A$ ($\mathrm{FV}(A) = \mathrm{FV}(t) = \emptyset$) and Contexts: $\Gamma ::= \cdot \mid \Gamma, A$
- Judgments: Σ **sig**, $\Sigma \vdash \Gamma$ **ctx**, $\Sigma; \Gamma \vdash A$ **type**, $\Sigma; \Gamma \vdash t : A$, $\Sigma; \Gamma \vdash A \equiv B$ **type**, $\Sigma; \Gamma \vdash t \equiv u : A$.
- For simplicity, $\Sigma; \Gamma \vdash \mathrm{Set} : \mathrm{Set}$.

Some examples

(We use variable names for clarity instead of deBruijn indices.)

Let $\Sigma_1 \stackrel{\text{def}}{=} \mathbb{A} : \mathrm{Set}, \mathfrak{a} : \mathbb{A}, \mathbb{b} : \mathbb{A}$.

- Σ_1 **sig**
- $\Sigma_1; \cdot \vdash \mathbb{A} : \mathrm{Set}$
- $\Sigma_1; \cdot \vdash \mathbb{A}$ **type**
- $\Sigma_1; \cdot \vdash \mathfrak{a} : \mathbb{A}$
-
- $\Sigma_1; \cdot \vdash \mathrm{if}\ (\lambda.\mathbb{A})\ \mathrm{true}\ \mathfrak{a}\ \mathbb{b} \equiv \mathfrak{a} : \mathbb{A}$

Let $\Sigma_2 \stackrel{\text{def}}{=} \mathbb{A} : \mathrm{Set}, \mathbb{b} : \mathbb{A}, \alpha := \lambda.\mathbb{b} : \mathrm{Bool} \to \mathbb{A}$.

- Σ_2 **sig**
- $\Sigma_2; \cdot \vdash \alpha : \mathrm{Bool} \to \mathbb{A}$
- $\Sigma_2; x : \mathrm{Bool} \vdash \alpha\, x \equiv \mathbb{b} : \mathbb{A}$

Pointed set:

- $\cdot; \Sigma(\mathrm{Set})(0) \vdash 0\ .\pi_2 : 0\ .\pi_1$
- $\cdot; x : \Sigma(g : \mathrm{Set})(g) \vdash x\ .\pi_2 : x\ .\pi_1$

Some examples

(We use variable names for clarity instead of deBruijn indices.)

Let $\Sigma_1 \stackrel{\text{def}}{=} \mathbb{A} : \mathrm{Set}, \mathbb{a} : \mathbb{A}, \mathbb{b} : \mathbb{A}$.

- Σ_1 **sig**
- $\Sigma_1; \cdot \vdash \mathbb{A} : \mathrm{Set}$
- $\Sigma_1; \cdot \vdash \mathbb{A}$ **type**
- $\Sigma_1; \cdot \vdash \mathbb{a} : \mathbb{A}$
-
- $\Sigma_1; \cdot \vdash \mathrm{if}\ (\lambda.\mathbb{A})\ \mathrm{true}\ \mathbb{a}\ \mathbb{b} \equiv \mathbb{a} : \mathbb{A}$

Let $\Sigma_2 \stackrel{\text{def}}{=} \mathbb{A} : \mathrm{Set}, \mathbb{b} : \mathbb{A}, \alpha := \lambda.\mathbb{b} : \mathrm{Bool} \to \mathbb{A}$.

- Σ_2 **sig**
- $\Sigma_2; \cdot \vdash \alpha : \mathrm{Bool} \to \mathbb{A}$
- $\Sigma_2; x : \mathrm{Bool} \vdash \alpha\, x \equiv \mathbb{b} : \mathbb{A}$

Pointed set:

- $\cdot; \Sigma(\mathrm{Set})(0) \vdash 0\ .\pi_2 : 0\ .\pi_1$
- $\cdot; x : \Sigma(g : \mathrm{Set})(g) \vdash x\ .\pi_2 : x\ .\pi_1$