Elaboration on HM(X): Type Inference with Constraint Types

Marius Weidner

Chair of Programming Languages, University of Freiburg
`weidner@cs.uni-freiburg.de`

Abstract. We explore HM(X)[5], a family of type systems designed to combine polymorphism, full type inference and constraint types. HM(X) extends the Hindley-Milner type system (HM)[4], which already limits System F, to ensure decidability and unambiguity of full type inference. The constraint system X utilized in HM(X) remains abstract, allowing instantiating X with arbitrary constraint systems that meet specific criteria. This abstraction allows HM(X) to serve as a model for analyzing various constraint-related type features commonly encountered in practice. Notable examples include subtyping, substructural types and type classes. HM(X) is presented along a sound and complete type inference algorithm that remains independent of the actual constraint system X. As a result, the work for proving theoretical properties and designing inference algorithms for novel constraint-based type systems within an HM context is notably simplified.

Table of Contents

1 Introduction

1.1 Polymorphism and Type Inference in HM

The HM type system represents a well known and understood typing discipline that refines System F by establishing constraints that allow type inference to be decidable. HM serves as the foundation for numerous real-world functional programming languages, including Haskell and Rust.

In System F, we can introduce variables for both expressions and types. The type $\forall\alpha.\ T$, where α binds a new type in T, indicates that an expression of this type is polymorphic over some arbitrary type α. Unfortunately it is undecidable for arbitrary programs to determine when to introduce and eliminate $\forall$-types and in consequence type inference in System F is undecidable[13].

Thus, System F is equipped with explicit type abstraction ($\Lambda\alpha.\ e$) and type application ($e\ T$) on the syntax level.

The HM type system imposes several restrictions that make type inference decidable in a polymorphic context. Moreover, HM ensures that the most general type (the *principal type*) of a any given program is inferred. Consequently, there's no need for extra syntax to introduce or eliminate type variables. Instead, HM adds let bindings **let** $x = e_2$ **in** e_1 to the language, where e_2 is the only expression that is allowed to have a polymorphic type. Other constructs, such as application or variables bound by a lambda abstractions, cannot inherit polymorphic types. This constraint is commonly referred to as 'let polymorphism'.

In the future, we will label polymorphic types as σ, where a $\forall$-type exists within σ, and we will refer to them as 'poly types'. On the contrary, all other types, including base types, functions and type variables, will be referred to as 'mono types' and denoted as τ.

Poly types are further constrained to exclusively permit $\forall$-binders at the top level. Therefore, a type like $(\forall\alpha.\alpha) \to (\forall\alpha.\alpha)$ would not be a valid poly type. Consequently, we establish that poly types always adhere to the structure $\forall\bar{\alpha}.\ \tau$, where $\bar{\alpha} = \alpha_1, .., \alpha_n$ for some $n \in \mathbb{N}$. By upholding these two restrictions, that is let polymorphism and the exclusion of higher-order polymorphism, type inference remains decidable and yields a principal type.

Example 1 (Concatination of Lists).

```
concat : ∀α. [α] → [α] → [α]
concat = λ[].      λys. ys
         λ[x:xs]. λys. x : (concat xs ys)
```

Examples assume the extension with various language features such as lists and pattern matching. For convenience, type annotations are given for inferred function types.

In this example, HM would be capable of deducing the type $\forall\alpha.\ [\alpha] \to [\alpha] \to [\alpha]$ for the function 'concat'. This type is in fact the most general type for 'concat'.

1.2 Introducing Constraints on Types

Although parametric polymorphism is already a powerful abstraction, there are instances where we desire to constrain type variables solely to instantiations of types that satisfy certain constraints. We refer to such types as *constraint types*. Constraints can exhibit various different forms depending on the type features present in the actual language. As an illustration, consider HM($\mathcal{R}$), that is HM extended with polymorphic records[7]. In this scenario, it becomes valuable to have the capability to specify that a type variable α should only be instantiated as a record type containing a specific fields.

Example 2 (Alternative to Selector Syntax on Records for Field 'key').

```
key : ∀α,β.(α ≤ {key : β}) ⇒ α → β
key = λ{key, ..}. key
```

This example simulates the extraction of a particular field from a record using pattern matching, conventionally represented as $e.l$ where l denotes a field label. Instead of the notation $e.key$, the alternative syntax $key\ e$ could then be employed. We introduce two type variables, namely α and β. While β exhibits parametric polymorphism, α functions as a constraint type. The constraint imposed on α, denoted as $\alpha \leq \{\text{key} : \beta\}$, expresses that α is exclusively permitted to take on a type that corresponds to a record featuring a 'key' field of type β. For introducing constraint a C on type variables $\bar{\alpha}$, we will adopt the notation $\forall \bar{\alpha}.C \Rightarrow \tau$. Multiple constraints will be combined to a single constraint using conjunction. If the constraint C is the trivial constraint $\top$ we may also write $\forall \bar{\alpha}.\ \tau$.

Naturally, we can envision entirely different constraints as well. Consider a language with overloading and overloading constraints HM($\mathcal{O}$)[6]. In HM($\mathcal{O}$), constraints have the form $x : \alpha \to \tau$, wherein an instance for the overloaded identifier x with type $\alpha \to \tau$ is expected to be present.

Example 3 (Overloading the Equality Operator for Lists).

```
eq : nat → nat → bool
eq = λ0.      λ0.      = ⊤
     λsuc n.  λsuc m.  = eq n m
     λ_.      λ_.      = ⊥
eq : ∀α.(eq : α → α → bool) ⇒ [α] → [α] → bool
eq = λ[].      λ[].      = ⊤
     λ[x:xs].  λ[y:ys].  = eq x y ∧ eq xs ys
     λ_.       λ_.       = ⊥
```

This example considers the overloaded 'eq' function. Initially, 'eq' is instantiated for the base type 'nat'. Subsequently, the instantiation for lists requires us to express that list equality is feasible only when the elements of the list can be compared, that is, there exists a instance $\alpha \to \alpha \to$ bool.

$$
\begin{array}{lll}
e ::= x & C ::= \top & \tau ::= \alpha \\
\quad \mid \lambda x.e & \quad \mid C \wedge C & \quad \mid \tau \to \tau \\
\quad \mid e\ e & \quad \mid \tau = \tau & \\
\quad \mid \textbf{let}\ x = e\ \textbf{in}\ e & \quad \mid \exists \alpha.\ C & \sigma ::= \tau \\
 & & \quad \mid \forall \vec{\alpha}.C \Rightarrow \tau \\
 & & \\
 & & \Gamma ::= \emptyset \\
 & & \quad \mid \Gamma,\ x : \sigma
\end{array}
$$

Fig. 1: Syntax

Instead of focusing exclusively on these individual systems, our exploration will center on HM(X), a HM-based system that remains detached from the actual constraint domain X. Subsequently, we will proceed to instantiate X to a constraint system $\boldsymbol{\mathcal{R}}$ that supports polymorphic records, similar to HM($\mathcal{R}$).

2 HM(X)

2.1 Introduction

In this segment, our discussion of HM(X) will adopt a slightly informal approach. We will skip over some specifics and formalities to ensure clarity and concentrate on grasping the underlying concepts of the system, rather than deriving a full formal definition.

2.2 Syntax & Constraints

The syntax of HM(X) is closely related to that of HM.

Expressions e include the constructs found in the simply typed lambda calculus, more specifically variables x, abstractions $\lambda x.\ e$, and applications $e\ e$. Additionally, let bindings are present to constrain the language to let polymorphism.

We maintain the distinction between mono types τ and poly types σ. However, we are now able to introduce constraints C in $\forall$-types.

Typing contexts Γ are either the empty context $\emptyset$ or an extension of another context with the type τ of a variable x, denoted as $\Gamma,\ x : \sigma$.

The constraint syntax described here forms the *minimal* essential components necessary for the syntax of constraint domain X. The underlying notion is that the constraint syntax is later extended when instantiating HM(X), and consequently also the syntax for expressions and types, to incorporate those new constraints.

A constraint C is either $\top$ (true), a conjunction of two constraints $C \wedge C$, an equality between mono types $\tau = \tau$, or a projection $\exists \alpha.\ C$. The projection

$$\frac{C,\ x:\sigma \in \Gamma}{C,\ \Gamma \vdash x:\sigma}\ \text{(Var)} \qquad \frac{C,\ (\Gamma,\ x:\tau') \vdash e:\tau}{C,\ \Gamma \vdash \lambda x.e:\tau}\ \text{(Abs)}$$

$$\frac{C,\ \Gamma \vdash e_1:\tau \to \tau' \quad C,\ \Gamma \vdash e_2:\tau'}{C,\ \Gamma \vdash e_1\ e_2:\tau'}\ \text{(App)} \qquad \frac{C,\ \Gamma \vdash e:\sigma \quad C,\ (\Gamma,\ x:\sigma) \vdash e':\tau'}{C,\ \Gamma \vdash \textbf{let}\ x = e\ \textbf{in}\ e':\tau'}\ \text{(Let)}$$

$$\frac{C \wedge D, \Gamma \vdash e:\tau \quad \vec{\alpha} \notin \mathit{free}(C,\Gamma)}{C \wedge \exists\vec{\alpha}.D,\ \Gamma \vdash e:\forall\vec{\alpha}.D \Rightarrow \tau}\ (\forall\text{I}) \qquad \frac{C,\ \Gamma \vdash e:\forall\vec{\alpha}.D \Rightarrow \tau \quad C \vdash [\vec{\tau}/\vec{\alpha}]D}{C,\ \Gamma \vdash e:[\vec{\tau}/\vec{\alpha}]\tau}\ (\forall\text{E})$$

Fig. 2: Logical Type System $(C, \Gamma \vdash e : \sigma)$

operator introduces a new type variable α to constraint C. Through projection, it becomes possible to express the existence of a type without the necessity of actually introducing a new type variable at the type level. This mechanism proves advantageous within both the non-algorithmic typing rules and type inference algorithms. Moreover, it actually extends the expressive capacity beyond that of solely introducing all type variables present in constraints using the $\forall$-operator.

We use the notation $C \vdash D$ to indicate that we are verifying that some constraint C implies another constraint D. Both the projection and the equality operator must satisfy several straightforward conditions, such as $C \vdash \exists\vec{\alpha}.\ C$ or $(\tau_1 = \tau_2) \vdash (\tau[\tau_1] = \tau[\tau_2])$, where both $\tau[\tau_1]$ and $\tau[\tau_2]$ substitute τ_1 and τ_2 at the same position in τ. We won't cover all the other, mostly trivial conditions.

2.3 Typing

Our current emphasis will be on the logical type system. The typing relation $(C, \Gamma \vdash e : \sigma)$ in HM(X) extends the standard typing relation $(\Gamma \vdash e : T)$ by integrating the propagation of a constraint C across the typing rules. The constraint C must be in a solved form. Informally, this means that C should be a satisfiable constraint and it should not contain equality predicates. Equality constraints should be resolved through some form of unification.

Unlike the inference algorithm, which is also provided for HM(X), the logical type system does not represent a determinable process for typing a provided expression. This is due to the $(\forall\text{E})$ and $(\forall\text{I})$ rules lacking syntax direction and being applicable at various points in the typing derivation.

Both (Abs) and (App) are the familiar standards found in the simply typed lambda calculus and operate on mono types, thus enforcing let polymorphism. On the other hand, the (Var) rule allows us to retrieve a variable with a poly type, potentially introduced through a let binding. The (Let) rule likewise enforces let polymorphism by permitting the expression bound by x to have a poly type, even though the resulting overall type remains monomorphic.

The $(\forall\text{I})$ and $(\forall\text{E})$ rules are particularly interesting as they enable the introduction and elimination of type variable bindings and constraints. Through the use of the $(\forall\text{I})$ rule, we can assign an arbitrary expression e, with current type τ, a poly type $\forall\vec{\alpha}.D \Rightarrow \tau$, introducing new type variables $\vec{\alpha}$ and the constraint D. Hence, we check that expression e has type τ under the assumption of $C \wedge D$.

Additionally, we add the constraint $\exists\bar{\alpha}.D$ in the conclusion of the rule using conjunction. This addition is not strictly necessary but expresses that D should be, at the very least, satisfiable, thereby enabling the detection of a type error arising from an unsatisfiable constraint at an earlier stage.

Interestingly, there are alternative approaches to handling D in the conclusion. For instance, one could entirely eliminate the satisfiability check [3] or opt for a less restricted[1] or more restricted satisfiability check[10]. However, the solution presented in HM(X) aligns most closely with our intuition, asserting that it is indeed true that D must be at least satisfiable for some $\bar{\alpha}$. One of the strengths of HM(X) lies in the introduction of the projection operator, a novel idea not found in the other variants.

The ($\forall$E) rule allows us to eliminate a $\forall$-type by substituting all type variables $\bar{\alpha}$ with mono types $\bar{\tau}$ in τ. This substitution of all bound variables with mono types results in a mono type again. More importantly we need to verify that the constraint D introduced by the $\forall$-type is satisfied when we substitute all $\bar{\alpha}$ by $\bar{\tau}$ in D, all while assuming the propagated constraint C.[1]

2.4 Type Inference

Type inference in HM(X) consists of two sequential steps. Initially, a typing problem is transformed into a constraint D within the constraint system X. Subsequently, constraint D undergoes normalization, a process involving the computation of a substitution σ and a constraint C in solved form.

Normalization guarantees that σC (where σC is the application of substitution σ to constraint C) implies D and that σC is a constraint in solved form. To ensure that a typing problem has the principal type property, it's necessary that constraints within X exhibit the most general normalizers. Analogous to types, we require that X must have the *principal constraint* property. This property ensures that the most general constraint results from normalization. Informally, this means for every constraint $D \in X$ we can compute a normalized constraint in solved form σC where σ is the minimal substitution necessary to transform D into a solved form constraint σC and C implies D. In the event that the constraint domain is the trivial constraint domain $\top$, the normalization procedure corresponds to the computation of the most general unifier within the Algorithm W employed in HM.

The actual inference algorithm takes four inputs: A substitution σ, A constraint in solved form C, Γ, and e. We write $\sigma, C, \Gamma \vdash e : \tau$ when we use the inference algorithm to deduce type τ for e. Initially, the first three are empty or trivially satisfied respectively, while e represents the expression for which we are deducing the type. Within the algorithm, we perform a case split on expressions, all the while ensuring that recursive calls maintain the invariant that σC remains

[1] The action of substitution on constraints can again be encoded *as constraint.* We have $[\vec{\tau}/\vec{\alpha}](\bigwedge_{i\in\mathbb{N}} C_i) \equiv \bigwedge_{i\in\mathbb{N}}([\vec{\tau}/\vec{\alpha}]C_i) \equiv \bigwedge_{i\in\mathbb{N}}(\exists\bar{\alpha}.C_i \wedge \bigwedge_j(\tau_j = \alpha_j))$. Moreover, we can extend this notion for arbitrary predicates P that might be part of the constraint language, i.e. $[\vec{\tau}/\vec{\alpha}]P \equiv \exists\bar{\alpha}.P \wedge \bigwedge_j(\tau_j = \alpha_j)$.

$$\begin{array}{lll} e \ ::= \ \ldots & C \ ::= \ \ldots & \tau \ ::= \ \ldots \\ \quad \mid \{l_i = e_i\} \text{ for } i \in \mathbb{N} & \quad \mid \tau \leq \{l : \tau\} & \quad \mid \{l_i : \tau_i\} \text{ for } i \in \mathbb{N} \\ \quad \mid e.l & & \end{array}$$

$$\frac{C,\ \Gamma \vdash e_i : \tau_i}{C,\ \Gamma \vdash \{l_i = e_i\} : \{l_i : \tau_i\}}\ (\text{Rec}) \qquad \frac{C,\ \Gamma \vdash e : \{..,l : \tau,..\}}{C,\ \Gamma \vdash e.l : \tau}\ (\text{Sel})$$

Fig. 3: HM($\mathcal{R}$) Extensions

a principal constraint by normalization. The resulting type is then obtained by applying σ to τ.

3 Instantiating HM(X)

3.1 HM($\mathcal{R}$): Instantiation with Polymorphic Records

We will now instantiate X to $\boldsymbol{\mathcal{R}}$, thus extending HM with polymorphic records and related constraints. Our focus will be only on the additions to the syntax, typing rules and minimal constraint language provided by HM(X).

The extensions introduce record expressions and types to the language, along with the record subtyping constraint $\tau \leq \{l : \tau'\}$, which says that τ is a record and must have at least the field l of type τ'. Moreover, a record $\{l_i = e_i\}$ has type $\{l_i : \tau_i\}$ only if each e_i is of type τ_i. Additionally, selecting a label l from e is assigned type τ if e is a record with a field l of type τ.

As mentioned, our desire is for the constraint domain $\boldsymbol{\mathcal{R}}$ to possess the principal constraint property. Initially, we impose some conditions on record subtyping constraints and subsequently examine when it is in solved form. We will not discuss some straightforward conditions of record constraints, such as the fact that every record trivially satisfies the subtyping constraint if it contains the corresponding field of the correct type, i.e. $\vdash \{..,l : \tau,..\} \leq \{l : \tau\}$.

We require that the order of labels is irrelevant. Formally, this implies that $\vdash \{l_i = e_i\} = \{l_{\pi(i)} = e_{\pi(i)}\}$, where π represents a permutation. Additionally, we disallow recursive constraints in the form of $\alpha \leq \{l : \tau\}$ where τ depends on the type variable α. The condition $\alpha \leq \{l : \tau\}$ also is the only subtyping record in solved form. The constraint system $\boldsymbol{\mathcal{R}}$ indeed satisfies the principal constraint property, and as a result, HM(X) grants us a sound and complete type inference algorithm for our system. It also provides a proof of the soundness of HM($\mathcal{R}$) itself.

Example 4 (Using the Projection Operator to Unpack Nested Records).

$\texttt{unpack : } \forall\alpha\exists\beta.(\alpha \leq \{\texttt{foo : }\beta\} \wedge \beta \leq \{\texttt{bar : nat}\}) \Rightarrow \alpha \rightarrow \texttt{ nat}$
$\texttt{unpack = } \lambda\{\texttt{foo : }\{\texttt{bar},..\},..\}\texttt{. bar}$

In this example, we use the projection operator to extract a field from a nested record. The use of projection, rather than introducing a second type variable for β, is chosen because we only refer to β in the constraint and not in the type signature.

4 Metatheory

4.1 Introduction

We will now look into some metatheoretical properties of both the logical type system and the inference algorithm presented in HM(X). In this discussion, we will once again adopt an informal approach, skipping some specifics, and omitting the proofs entirely.

4.2 Soundness

The logical type system of HM(X) is proven to be sound. Soundness essentially states the idea that 'well typed programs don't go wrong'. In the original paper, this proof is established through denotational semantics, where mathematical objects are constructed for both types and terms, and the statements are proven within the domain of mathematics. Later, it was also proven using operational semantics, where evaluation is described as a reduction relation $e \hookrightarrow e$ on expressions[9]. We will concentrate on the latter, where the soundness property is established by ensuring that the language satisfies both *progress* and *subject reduction*. The actual paper also includes the modeling of memory, so the statements proven there may vary slightly from the ones discussed here.

First and foremost, we must ensure that the constraint system is sound. The constraint system is sound if, for every constraint in solved form, $\vdash \exists\alpha.\ C$ implies that there exists some type τ such that $\vdash [\alpha/\tau]C \equiv \vdash \exists\alpha.\ (\alpha = \tau) \wedge C$. In essence, this implies that every existentially quantified constraint must be actually satisfiable for some type τ.

To establish progress for HM(X), we must prove that if e is a well-typed expression, meaning $C, \emptyset \vdash e : \tau$, then either e is already a fully reduced value or it can be reduced to some other expression, i.e. $e \hookrightarrow e'$. Since this statement only holds in the empty context $\emptyset$, we need to ensure that there is no evaluation of expressions where non-substituted variables still exist. This can be achieved using call-by-value semantics.

Finally, to establish subject reduction, we need to show that for every well-typed expression $C, \Gamma \vdash e : \tau$, if $e \hookrightarrow e'$ then e' is also well-typed, specifically with the same type, i.e. $C, \Gamma \vdash e' : \tau$.

4.3 Type Inference

The soundness and completeness properties of the inference algorithm are a bit more challenging to articulate since we discussed the inference algorithm only informally. We will only discuss the key ideas.

HM(X) establishes soundness for the inference algorithm by relating the algorithm to the logical type system. We prove that if $\sigma, C, \Gamma \vdash e : \tau$, then $C, \Gamma \vdash e : \tau$. In essence, we check that if the algorithm deduces the type τ for e, then e has type τ in the logical type system.

Completeness, in essence, is the reverse. We must show that if the logical type system deduces τ for e, then the algorithm would also infer the same type τ for e.

5 Related Work & Conclusion

5.1 Related Work

There are several constraint-based languages based on HM, such as HM($\mathcal{R}$)[7] and HM($\mathcal{O}$)[6]. Additionally, there are alternative approaches to imposing arbitrary constraint systems on types, for example the theory of qualified types[2].

HM(X) is also the subject of various follow-up papers, including proofs of other noteworthy properties[11], its connection to CSP solving[12] and the previously discussed syntactic type soundness[9].

5.2 Conclusion

We discussed HM(X), a language based on HM that incorporates an abstract constraint system X. We outlined the extensions that HM(X) introduces in contrast to HM, providing an informal understanding of the core concepts of HM(X) and the role of X. Furthermore, we instantiated X with a specific constraint domain $\boldsymbol{\mathcal{R}}$. Lastly, we explored the metatheoretical properties of HM(X). These properties hold particular significance because of their independence from the actual constraint domain X.

References

[1] Alexander Aiken and Edward L. Wimmers. "Type Inclusion Constraints and Type Inference". In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. FPCA '93. Copenhagen, Denmark: Association for Computing Machinery, 1993, pp. 31–41. ISBN: 089791595X. DOI: `10.1145/165180.165188`. URL: `https://doi.org/10.1145/165180.165188`.

[2] Mark P. Jones. "A theory of qualified types". In: *ESOP '92*. Ed. by Bernd Krieg-Brückner. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 287–306. ISBN: 978-3-540-46803-5.

[3] Mark P. Jones. *Qualified Types: Theory and Practice*. USA: Cambridge University Press, 1995. ISBN: 0521472539.

[4] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. ISSN:

0022-0000. DOI: `https://doi.org/10.1016/0022-0000(78)90014-4`. URL: `https://www.sciencedirect.com/science/article/pii/0022000078900144`.

[5] Martin Odersky, Martin Sulzmann, and Martin Wehr. "Type Inference with Constrained Types". In: *TAPOS* 5 (Jan. 1999), pp. 35–55. DOI: `10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4`.

[6] Martin Odersky, Philip Wadler, and Martin Wehr. "A Second Look at Overloading". In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. FPCA '95. La Jolla, California, USA: Association for Computing Machinery, 1995, pp. 135–146. ISBN: 0897917197. DOI: `10.1145/224164.224195`. URL: `https://doi.org/10.1145/224164.224195`.

[7] Atsushi Ohori. "A Polymorphic Record Calculus and Its Compilation". In: *ACM Trans. Program. Lang. Syst.* 17.6 (Nov. 1995), pp. 844–895. ISSN: 0164-0925. DOI: `10.1145/218570.218572`. URL: `https://doi.org/10.1145/218570.218572`.

[8] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. ISBN: 0262162288.

[9] Christian Skalka and François Pottier. "Syntactic Type Soundness for HM(X)". In: *Electronic Notes in Theoretical Computer Science* 75 (2003). TIP'02, International Workshop in Types in Programming, pp. 61–74. ISSN: 1571-0661. DOI: `https://doi.org/10.1016/S1571-0661(04)80779-5`. URL: `https://www.sciencedirect.com/science/article/pii/S1571066104807795`.

[10] Geoffrey Smith. "Polymorphic type inference for languages with overloading and subtyping /". In: (Jan. 1991).

[11] Martin Sulzmann. "Proofs of Properties about HM(X)". In: 1998. URL: `https://api.semanticscholar.org/CorpusID:7951782`.

[12] MARTIN SULZMANN and PETER J. STUCKEY. "HM(X) type inference is CLP(X) solving". In: *Journal of Functional Programming* 18.2 (2008), pp. 251–283. DOI: `10.1017/S0956796807006569`.

[13] B. Wells J. *Typability and type checking in the second-order Λ-calculus are equivalent and undecidable (Preliminary Draft)*. 1993. URL: `https://open.bu.edu/handle/2144/1474`.