Gradual Typing with Unification-based Inference

Jeremy G. Siek
Dept. of Electrical and Computer Engineering
University of Colorado at Boulder
425 UCB
Boulder, CO 80309 USA

Manish Vachharajani
Dept. of Electrical and Computer Engineering
University of Colorado at Boulder
425 UCB
Boulder, CO 80309 USA

CU Technical Report CU-CS-1039-08
January 2008

Abstract

Static and dynamic type systems have well-known strengths and weaknesses. *Gradual typing* provides the benefits of both in a single language by giving the programmer control over which portions of the program are statically checked based on the presence or absence of type annotations.

This paper studies the combination of gradual typing and unification-based type inference, with the goal of developing a system that helps programmers increase the amount of static checking in their program. The key question in combining gradual typing and inference is how should the dynamic type of a gradual system interact with the type variables of an inference system. This paper explores the design space and shows why three straightforward approaches fail to meet our design goals. In particular, the combined system should satisfy the criteria for a gradual type system: 1) when a program is unannotated, only a few type errors are detected at compile-time and the rest are detected at run-time, and 2) when the program does not contain dynamic type annotations (implicitly or explicitly), the type system should statically detect all type errors.

This paper presents a new type system based on the idea that a solution for a type variable should be as informative as any type that constrains the variable. We prove that the new type system satisfies the above criteria for a gradual type system. The paper also develops an efficient inference algorithm and proves it sound and complete with respect to the type system.

Contents

1 Introduction

Static and dynamic typing have complementary strengths, making them better for different tasks and stages of development. Static typing, used in languages such as Standard ML [18], provides full-coverage type error detection, facilitates efficient execution (since values may remain unboxed and run-time checking of type tags is not needed), and provides machine-checked documentation that is particularly helpful for maintaining consistency when programming in the large. The main drawback of static typing is that the whole program must be well-typed before the program can be run. Typing decisions must be made for all elements of the program, even for ones that have yet to stabilize, and changes in these elements can ripple throughout the program.

In a dynamically typed language, no compile-time checking is performed. Thus, programmers need not worry about types while the overall structure of the program is still in flux. This makes dynamic languages suitable for rapid prototyping. Dynamically typed languages such as Perl, Ruby, Python, and JavaScript are popular for scripting and web applications where rapid development and prototyping is prized above other features. The problem with dynamic languages is that they forgo the benefits of static typing: there is no machine checked documentation, execution is less efficient, and errors are caught only at runtime, often after deployment.

Gradual typing, recently introduced by Siek and Taha [29], enable programmers to mix static and dynamic type checking in a program by providing a convenient way to control which parts of a program are statically checked. The defining properties of a gradual type system are:

1. Programmers may omit type annotations and immediately run the program; run-time type checks are performed to preserve type safety.

2. Programmers may add type annotations to increase static checking. When all variables are annotated, *all* type errors are caught at compile-time.

A number of researchers have further studied gradual typing over the last two years. Herman, Tomb, and Flanagan [12] developed space-efficient run-time support for gradual typing. Siek and Taha [30] integrated gradual typing with objects and subtyping. Wadler and Findler showed how to perform blame tracking and proved that the well-typed portions of a program can't be blamed [36]. Herman and Flanagan are adding gradual typing to the next version of JavaScript [11].

An important question, from both a theoretical and practical perspective, has yet to be answered: is gradual typing compatible with type inference? Type inference is common in modern functional languages and is becoming more common in mainstream languages [10, 37]. There are many flavors of type inference: Hindley-Milner inference [17], dataflow-based inference [6], Soft Typing [3], and local inference [24] to name a few. In this paper we study type inference based on unification [28], the foundation of Hindley-Milner inference and the related family of algorithms used in many functional languages [16, 18, 22].

The contributions of this paper are:

1. An exploration of the design space that shows why three straightforward inference approaches do not satisfy the above criteria for a gradual type system (§3). The three approaches are: 1) treat dynamic types as type variables, 2) well-typed after substitution, and 3) ignore dynamic types during unification.

2. A new design based on the idea that the solution for a type variable should be as informative as any type that constrains the variable (§4). We formalize this idea in a type

system (§4.2) and prove that it satisfies the criteria of a gradual type system (§4.3). We machine checked the proofs in Isabelle/HOL [20]. The formalization and proofs are in Appendix A.

3. An inference algorithm for the above type system (§5). We prove that the algorithm is sound and complete with respect to the type system and that the algorithm has almost linear time complexity (§5.3). The algorithm does not infer types that introduce unnecessary cast errors.

Before the main technical developments, we review gradual typing as well as traditional unification-based inference (§2). After the main body of the paper, we place our work in relation to the relevant literature (§6) and conclude (§7).

2 Review of Gradual Typing and Inference

We review gradual typing in the absence of type inference, showing examples in a hypothetical variant of Objective Caml [16] that supports gradual typing but not type inference. We then review type inference in the absence of gradual typing.

A Review of Gradual Typing The incr function listed below has a parameter x and returns x + 1. The parameter x does not have a type annotation so the gradual type system delays checks concerning x inside the incr function until run-time, just as a dynamically typed language would.

```
let incr x = x + 1
let a:int = 1
incr a
```

More precisely, because the parameter x is not annotated the gradual type system gives it the **dynamic type**, written ? for short.

Now suppose the + operator expects arguments of type int. The gradual type system allows an implicit coercion from type ? to int. This kind of coercion could fail (like a down cast) and therefore must be dynamically checked. In some statically-typed languages, such as ML, implicit coercions are forbidden; in many object-oriented languages, such as Java, implicit up-casts are allowed (they never fail) but not implicit down-casts. Allowing implicit coercions that may fail is *the* distinguishing feature of gradual typing and gives it the flavor of dynamic typing.

To facilitate the migration of code from dynamic to static checking, gradual typing allows for a mixture of the two and provides seamless interaction between them. In the example above, we define a variable a of type int, and invoke the dynamically typed incr function. Here the gradual type system allows an implicit coercion from int to ?. This is a safe coercion—it can never fail at run-time—however the run-time system needs to remember the type of the value so that it can check the type when it casts back to int inside of incr.

Gradual typing also allows implicit coercions among more complicated types, such as function types. In the following example, the map function has a parameter f annotated with the function type (int → int) and a parameter l with type int list.

```
let rec map (f:int→int) (l:int list) = ...
let incr x = x + 1
let a:int = 1
map incr [1; 2; 3] (* OK *)
map a [1; 2; 3] (* compile time type error *)
```

The function call `map incr [1; 2; 3]` is allowed by the gradual type system, even though the type of the argument `incr` ($? \to \mathsf{int}$) differs from the type of the parameter ($\mathsf{int} \to \mathsf{int}$). The type system compares the two types structurally and allows the two types to differ in places where one of the types has a ?. Thus, the function call is allowed because the return types are equal and there is a ? in one of the parameter types. In contrast, `map a [1; 2; 3]` elicits a compile-time error because argument `a` has type `int` whereas `f` is annotated with a function type.

When a program is fully annotated, that is, when all the program variables are annotated with types that include no ? types, the gradual type system catches at compile-time all the errors that a fully-static type system would.

More formally, the main idea of gradual typing is to replace the use of type equality with a relation called type consistency, written $\sim$ for short. The intuition behind type consistency is to check whether the two types are equal in the parts where both types are known. The following are a few examples.

$$
\begin{array}{cccc}
\mathsf{int} \sim \mathsf{int} & \mathsf{int} \not\sim \mathsf{bool} & ? \sim \mathsf{int} & \mathsf{int} \sim ? \\
\multicolumn{2}{c}{\mathsf{int} \to ? \sim ? \to \mathsf{int}} & \multicolumn{2}{c}{\mathsf{int} \to ? \sim \mathsf{int} \to \mathsf{bool}} \\
\multicolumn{2}{c}{\mathsf{int} \to ? \not\sim \mathsf{bool} \to ?} & \multicolumn{2}{c}{\mathsf{int} \to \mathsf{int} \not\sim \mathsf{int} \to \mathsf{bool}}
\end{array}
$$

The following is an inductive definition of the consistency relation. This relation is reflexive, symmetric, but not transitive.

Type Consistency

$$
(\textsc{CRefl})\ \frac{}{\tau \sim \tau} \qquad (\textsc{CFun})\ \frac{\tau_1 \sim \tau_2 \qquad \rho_1 \sim \rho_2}{\tau_1 \to \rho_1 \sim \tau_2 \to \rho_2}
$$

$$
(\text{CDR})\ \frac{}{\tau \sim ?} \qquad (\text{CDL})\ \frac{}{? \sim \tau}
$$

The syntax of the gradually typed lambda calculus ($\lambda^{?}_{\to}$) is shown below and the type system is reproduced in Figure 1. A gradual type system uses type consistency where a simple type system uses type equality. For example, the ($\textsc{App}2$) rule in the gradually typed lambda calculus [29] requires that the argument type τ_2 be consistent with the parameter type τ_1.

Syntax for $\lambda^{?}_{\to}$

$$
\begin{array}{llll}
\text{Variables} & x, y & \in \mathbb{X} & \\
\text{Ground Types} & \gamma & \in \mathbb{G} & \supseteq \{\mathsf{bool}, \mathsf{int}, \mathsf{unit}\} \\
\text{Constants} & c & \in \mathbb{C} & \supseteq \{\mathsf{true}, \mathsf{false}, \mathsf{succ}, 0, (), \mathsf{fix}[\tau]\} \\
\text{Types} & \tau & ::= & ? \mid \gamma \mid \tau \to \tau \\
\text{Expressions} & e & ::= & x \mid c \mid e\ e \mid \lambda x\!:\!\tau.\ e \\
 & \lambda x.\ e & \equiv & \lambda x\!:\!?.\ e
\end{array}
$$

This type system meets both criteria for a gradual type system discussed in §1: the first because the consistency relation allows implicit coercions both to and from the dynamic type, the second because when there are no ?s in the program (either explicitly or implicitly), this type system is equivalent to a fully static type system. The consistency relation collapses to equality when there are no ?s: for any σ and τ that contain no ?s, $\sigma \sim \tau$ iff $\sigma = \tau$.

Review of Unification-based Type Inference Type inference allows programmers to omit type annotations but still enjoy the benefits of static type checking. For example, the following

$$\boxed{\Gamma \vdash_g e : \tau}$$

$$(\textsc{Var})\qquad \frac{\Gamma(x) = \tau_1}{\Gamma \vdash_g x : \tau_1}$$

$$(\textsc{Cnst})\qquad \Gamma \vdash_g c : \mathit{typeof}(c)$$

$$(\textsc{App1})\qquad \frac{\Gamma \vdash_g e_1 : ? \qquad \Gamma \vdash_g e_2 : \tau}{\Gamma \vdash_g e_1\ e_2 : ?}$$

$$(\textsc{App2})\qquad \frac{\begin{array}{c}\Gamma \vdash_g e_1 : \tau_1 \rightarrow \tau_3 \qquad \Gamma \vdash_g e_2 : \tau_2 \\ \tau_1 \sim \tau_2\end{array}}{\Gamma \vdash_g e_1\ e_2 : \tau_3}$$

$$(\textsc{Abs})\qquad \frac{\Gamma(x \mapsto \tau_1) \vdash_g e : \tau_2}{\Gamma \vdash_g \lambda x : \tau_1.\ e : \tau_1 \rightarrow \tau_2}$$

Figure 1: The type system for $\lambda^{?}_{\rightarrow}$.

is a well-typed Objective Caml program. The inference algorithm deduces that the type of function f is int → int.

```
# let f x = x + 1;;
val f : int →int = ⟨fun⟩ (* Output of inference *)
```

The type inference problem is formulated by attaching a type variable, an *unknown*, to each location in the program. The job of the inference algorithm is to deduce a solution for these variables that obeys the rules of the type system. So, for example, the following is the above program annotated with type variables.

$$\textbf{let}\ \mathsf{f}_\alpha\ \mathsf{x}_\beta = (\mathsf{x}_\gamma +_\delta 1_\chi)_\rho$$

The inference algorithm models the rules of a type system as equations that must hold between the type variables. For example, the type β of the parameter x must be equal to the type γ of the occurrence of x in the body of f. The parameter types of + (both are int) must be equal to the argument types γ and χ, and the return type of +, also int, must be equal to ρ. Ultimately, the type α of f must be equal to the function type $\beta \rightarrow \rho$ formed from the parameter type β and the return type ρ. This set of equations can be solved by standard unification [28]. A substitution maps type variables to types and can be naturally extended to map types to types. The unification algorithm computes a substitution S such that for each equation $\tau_1 = \tau_2$, we have $S(\tau_1) = S(\tau_2)$.

A natural setting in which to formalize type inference is the simply typed lambda calculus with type variables ($\lambda^{\alpha}_{\rightarrow}$). The syntax is similar to $\lambda^{?}_{\rightarrow}$, but with type variables and no dynamic type. The standard type system for the simply typed lambda calculus [23] is reproduced in Figure 2. The extension of this type system to handle type variables, given below, is also standard [23].

Definition 1. *A term e of $\lambda^{\alpha}_{\rightarrow}$ is* ***well-typed*** *in environment Γ if there is a substitution S and a type τ such that $S(\Gamma) \vdash S(e) : \tau$.*

We refer to this approach to defining well-typedness for programs with type variables as *well-typed after substitution.*

An inference algorithm for $\lambda^{\alpha}_{\rightarrow}$ can be expressed as a two-step process [8, 23, 38] that generates a set of constraints (type equalities) from the program and then solves the set of equalities with unification. Constraint generation for $\lambda^{\alpha}_{\rightarrow}$ is defined in Figure 3. The soundness and completeness of the inference algorithm with respect to the type system has been proved in the literature [23, 38].

$$\frac{\Gamma(x) = \tau_1}{\Gamma \vdash x : \tau_1} \qquad \boxed{\Gamma \vdash e : \tau}$$

$$\Gamma \vdash c : \mathit{typeof}(c)$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma(x \mapsto \tau_1) \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}$$

Figure 2: The type system of the simply typed λ-calculus.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \mid \{\}} \qquad \boxed{\Gamma \vdash e : \tau \mid C}$$

$$\Gamma \vdash c : \mathit{typeof}(c) \mid \{\}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \mid C_1 \\ \Gamma \vdash e_2 : \tau_2 \mid C_2 \qquad (\beta \text{ fresh})\end{array}}{\Gamma \vdash e_1\ e_2 : \beta \mid \{\tau_1 = \tau_2 \to \beta\} \cup C_1 \cup C_2}$$

$$\frac{\Gamma(x \mapsto \tau) \vdash e : \rho \mid C}{\Gamma \vdash \lambda x : \tau.\ e : \tau \to \rho \mid C}$$

Figure 3: The definition of constraint generation for $\lambda^{\alpha}_{\to}$.

In §4 we combine inference with gradual typing and need to treat type variables with special care, but if we follow the well-typed-after-substitution approach, type variables are substituted away before the type system is consulted. As an intermediate step towards integration with gradual typing, we give an equivalent definition of well-typed terms for $\lambda^{\alpha}_{\to}$ that combines the substitution S with the type system. The type system is shown in Figure 4 and the judgment has the form $S; \Gamma \vdash e : \tau$ which reads: e is well-typed because S and τ are a solution for e in Γ.

Formally, we use the following representation for substitutions, which is common in mechanized formalizations [19].

Definition 2. *A* ***substitution*** *is a total function from type variables to types and its* ***dom*** *consists of the variables that are not mapped to themselves. Substitutions extend naturally to types, typing environments, and expressions. The $\circ$ operator composes two substitutions.*

Theorem 1 states that the two type systems are equivalent, and relies on the following two lemmas. The function FTV returns the free type variables within a type, type environment, or expression.

Lemma 1. *If $S(\Gamma) \vdash S(e) : \tau$ and S is idempotent then $S(\tau) = \tau$.*

Proof. Observe that if $S(\Gamma) \vdash S(e) : \tau$ then $\mathrm{FTV}(\tau) \cap \mathrm{dom}(S) = \emptyset$. Furthermore, if S is idempotent then $\mathrm{FTV}(\tau) \cap \mathrm{dom}(S) = \emptyset$ implies $S(\tau) = \tau$. ☐

Lemma 2. *If S idempotent and $S(\tau) = \tau_1 \to \tau_2$ then $S(\tau_2) = \tau_2$.*

Proof. We have $\tau_1 \to \tau_2 = S(\tau) = S(S(\tau)) = S(\tau_1 \to \tau_2) = S(\tau_1) \to S(\tau_2)$. Thus $\tau_2 = S(\tau_2)$. ☐

$$\text{(S}\textsc{Var}) \qquad \frac{\Gamma(x) = \tau}{S;\Gamma \vdash x : \tau} \qquad\qquad \boxed{S;\Gamma \vdash e : \tau}$$

$$\text{(S}\textsc{Cnst}) \qquad S;\Gamma \vdash c : \mathit{typeof}(c)$$

$$\text{(S}\textsc{App}) \qquad \frac{S;\Gamma \vdash e_1 : \tau_1 \quad S;\Gamma \vdash e_2 : \tau_2 \quad S(\tau_1) = S(\tau_2 \to \tau_3)}{S;\Gamma \vdash e_1\ e_2 : \tau_3}$$

$$\text{(S}\textsc{Abs}) \qquad \frac{S;\Gamma(x \mapsto \tau_1) \vdash e : \tau_2}{S;\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}$$

Figure 4: The type system for $\lambda^{\alpha}_{\rightarrow}$.

Theorem 1. *The two type systems for* $\lambda^{\alpha}_{\rightarrow}$ *are equivalent.*

1. *Suppose* S *is idempotent. If* $S(\Gamma) \vdash S(e) : \tau$*, then there is a* τ' *such that* $S;\Gamma \vdash e : \tau'$ *and* $S(\tau') = \tau$.
2. *If* $S;\Gamma \vdash e : \tau$*, then* $S(\Gamma) \vdash S(e) : S(\tau)$.

Proof. 1. $S(\Gamma) \vdash S(e) : \tau \implies S(\Gamma) \vdash S(e) : S(\tau)$ by Lemma 1. We prove by induction that $S(\Gamma) \vdash S(e) : S(\tau)$ implies there is a τ' such that $S;\Gamma \vdash e : \tau'$ and $S(\tau') = S(\tau)$. We use Lemma 1 in the (App) case and Lemma 2 in the (Abs) case. Then using Lemma 1 once more gives us $S(\tau') = \tau$.

2. The proof is a straightforward induction on $S;\Gamma \vdash e : \tau$. ☐

3 Exploration of the Design Space

We investigate three straightforward approaches to integrate gradual typing and type inference. In each case we give examples of programs that should be well-typed but are rejected by the approach, or that should be ill-typed but are accepted by the approach.

Dynamic Types as Type Variables A simple approach is to replace every occurrence of ? in the program with a fresh type variable and then do constraint generation and unification as presented in §2. The resulting system is fully static, not gradual. Consider the following program.

```
let z = ...
let f (x : int) = ...
let g (y : bool) = ...
let h (a : ?) = if z then f a else g a
```

Variable `a` has type ? and so a fresh type variable α would be introduced for its type. The inference algorithm would deduce from the function applications `f a` and `g a` that $\alpha = \mathsf{int}$ and $\alpha = \mathsf{bool}$ respectively. There is no solution to these equations, so the program would be rejected with a static type error. However, the program would run without error in a dynamically typed language given an appropriate value of `z` and input for `h`. Furthermore, this program type checks in the gradual type system of Figure 1 so it ought to remain valid in the presence of type inference.

The next example exhibits a different problem: the inference algorithm may not find concrete solutions for some variables and therefore indicate polymorphism in cases where there shouldn't be.

```
let f (x : int) (g : ? →?) =
    g x
```

Generating fresh type variables for the ?s gives us g : $\alpha \rightarrow \beta$. Let γ be the type variable for the return type of f and the type of the expression g x. The only equation constraining γ is $\gamma = \beta$, so the return type of f is inferred to be β. But if f is really polymorphic in β it should behave uniformly for any choice β [27, 35]. Suppose g is the identity function. Then f raises a cast error if $\beta =$ bool but not if $\beta =$ int.

Ignore Dynamic Types During Unification Yet another straightforward approach is to adapt unification by simply ignoring any unification of the dynamic type with any other type. However, this results in programs with even more unsolved variables than in the approach described above. Consider again the following program.

```
let f (x : int) (g : ? →?) =
    g x
```

From the function application, the inference algorithm would deduce $? \rightarrow ? = \mathsf{int} \rightarrow \beta$, where β is a fresh variable representing the result type of the application g x. This equality would decompose to $? =$ int and $? = \beta$. However, if the unification algorithm does not do anything with $? = \beta$, we end up with β as an unsolved variable, giving the impression that f is parametric in β, which is certainly not the case. Some choices for β can cause runtime cast errors whereas other choices do not.

Well-typed After Substitution In §2 we presented the standard type system for $\lambda^{\alpha}_{\rightarrow}$, saying that a program is well typed if there is some substitution that makes the program well typed in $\lambda_{\rightarrow}$. We could do something similar for gradual typing, saying that a gradually typed program with variables is well typed if there exists a substitution that makes it well typed in $\lambda^{?}_{\rightarrow}$ (Figure 1).

It turns out that this approach is too lenient. Recall that to satisfy criteria 2 of gradual typing, for fully annotated programs the gradual type system should act like a static type system. Consider the following program that would not type check in a static type system because α cannot be both an int and a function.

```
let f (g:α) = g 1
f 1
```

Applying the substitution $\{\alpha \mapsto ?\}$ produces a program that is well-typed in $\lambda^{?}_{\rightarrow}$.

The next example shows a less severe problem, although it still undermines the purpose of type inference, which is to help programmers increase the amount of static typing in their programs.

```
let x:α = x + 1
```

Again, the substitution $\{\alpha \mapsto ?\}$ is allowed, but it does not help the programmer. Instead, one wants to find out that $\alpha =$ int. In general, we need to be more careful about where ? is allowed as the solution for a type variable.

However, we cannot altogether disallow the use of ? in solutions because we want to avoid introducing runtime cast errors. Consider the program

```
let f (x:?) =
    let y:α = x in y
```

Here, the *only* appropriate solution for α is the dynamic type. Any other choice introduces an implicit cast to that type, which causes a runtime cast error if the function is applied to a value whose type does not match our choice for α. Suppose we choose $\alpha = \mathsf{int}$. This type checks in $\lambda^{?}_{\rightarrow}$ because int is consistent with ?, but if the function is called with a boolean argument, a runtime cast error occurs.

The problem with the well-typed-after-substitution approach is that it can "cheat" by assigning ? to a type variable and thereby allow programs to type check that should not. Thus, we need to prevent the type system from adding in arbitrary ?s. On the other hand, we need to allow the propagation of ?s that are already in program annotations.

4 A Type System for $\lambda^{?\alpha}_{\rightarrow}$

Loosely, we say that types with more question marks are less informative. The main idea of our new type system is to require the solution for a type variable to be as informative as any type that constrains the type variable. This prevents a solution for a variable from introducing dynamic types that do not already appear in program annotations. Formally, information over types is characterized by the *less or equally informative* relation, written $\sqsubseteq$. This relation is just the partial order underlying the $\sim$ relation[1]. An inductive definition of $\sqsubseteq$ is given below.

Less or Equally Informative

$$\text{(LID)}\,\frac{}{?\sqsubseteq\tau}\qquad\text{(LIR}\textsc{efl}\text{)}\,\frac{}{\tau\sqsubseteq\tau}$$

$$\text{(LIF}\textsc{un}\text{)}\,\frac{\tau_1\sqsubseteq\tau_3\qquad\tau_2\sqsubseteq\tau_4}{\tau_1\rightarrow\tau_2\sqsubseteq\tau_3\rightarrow\tau_4}$$

The $\sqsubseteq$ relation is a partial order that forms a semi-lattice with ? as the bottom element and $\sqsubseteq$ extends naturally to substitutions.

We revisit some examples from §3 and show how using the $\sqsubseteq$ relation gives us the ability to separate the good programs and good solutions from the bad. Recall the following example that should be rejected but was not using the well-typed-after-substitution approach.

```
let f (g:α) = g 1
f 1
```

In our approach, the application of g to 1 introduces the constraint $\mathsf{int} \rightarrow \beta_0 \sqsubseteq \alpha$ (where β_0 is a fresh variable generated for the result of the application) because g is being used as a function from int to β_0. Likewise, the application of f to 1 introduces the constraint $\mathsf{int} \rightarrow \beta_1 \sqsubseteq \alpha \rightarrow \beta_0$ which implies $\mathsf{int} \sqsubseteq \alpha$. There is no solution to these constraints on α so the program is rejected.

In the next example, the only solution for α should be int.

```
let x:α = x + 1
```

Indeed, in our approach we have the constraint $\mathsf{int} \sqsubseteq \alpha$ whose only solution is $\alpha = \mathsf{int}$.

In the third example, the type system should allow $\alpha = ?$ as a solution.

---

[1] Each relation is definable in terms of the other: we have $\tau_1 \sim \tau_2$ iff there is a τ_3 such that $\tau_1 \sqsubseteq \tau_3$ and $\tau_2 \sqsubseteq \tau_3$, and in the other direction, $\tau_1 \sqsubseteq \tau_2$ iff for any τ_3, $\tau_2 \sim \tau_3$ implies $\tau_1 \sim \tau_3$.

```
let f (x:?) =
    let y:α = x in y
```

Indeed, we have the constraint $? \sqsubseteq \alpha$, which allows $\alpha = ?$ as a solution. In this case the type system allows many solutions, some of which, as discussed in §3 may introduce unnecessary casts. In our design, the inference algorithm is responsible for choosing a solution that does not introduce unnecessary casts. It will do this by choosing the least informative solution allowed by the type system. This means the inference algorithm chooses the least upper bound of all the types that constraint a type variable as the solution for that variable.

The following program further illustrates how the $\sqsubseteq$ relation constrains the set of valid solutions.

```
let f (g:?→int) (h:int→?) = ...
let k (y:α) = f y y
```

The parameter y is annotated with type variable α and is used in two places, one that expects $? \to \mathsf{int}$ and the other that expects $\mathsf{int} \to ?$. So we have the constraints $? \to \mathsf{int} \sqsubseteq \alpha$ and $\mathsf{int} \to ? \sqsubseteq \alpha$, whose only solution is $\alpha = \mathsf{int} \to \mathsf{int}$.

Constraints on type variables can also arise from constraints on compound types that contain type variables. For example, in the following program, we need to delve under the function type to uncover the constraint that $\mathsf{int} \sqsubseteq \alpha$.

```
let g (f:int→int) = f 1
let h (f:α→α) = g f
```

In the next subsection we define how this works in our type system.

4.1 The Consistent-equal and Consistent-less Judgments

To formalize the notions of constraints between arbitrary types, we introduce two judgments: consistent-equal, which has the form $S \models \tau \simeq \tau$ and consistent-less, which has the form $S \models \tau \sqsubseteq \tau$. The two judgments are defined in Figure 5. The consistent-equal judgment is similar to the type consistency relation $\sim$ except that $\simeq$ gives special treatment to variables. When a variable occurs on either side of the $\simeq$, the substitution for that variable is required to produce a type that is as informative as the other type according to the consistent-less judgment. The consistent-less judgment is similar to the $\sqsubseteq$ relation except that it also gives special treatment to variables. When a variable appears on the left, the substitution for that variable is required to be equal to the type on the right. (There is some asymmetry in the $S \models \tau \sqsubseteq \tau$ judgment. The substitution is applied to tyep of the left and not the right because the substitution has already been applied to the type on the right.)

We illustrate the rules for consistent-equal and consistent-less with the following example.

$$S \models \mathsf{int} \to \alpha \simeq\ ? \to (\beta \to (\mathsf{int} \to ?))$$

What choices for S satisfies the above constraint? Applying the inverse of the (CEFun) rule we have

$$S \models \mathsf{int} \simeq ?,\quad S \models \alpha \simeq \beta \to (\mathsf{int} \to ?)$$

The first constraint is satisfied by any substitution using rule (CEDR), but the second constraint is satisfied when

$$S \models \beta \to (\mathsf{int} \to ?) \sqsubseteq S(\alpha)$$

$$\text{(CEG)} \quad \frac{}{S \models \gamma \simeq \gamma} \qquad \boxed{S \models \tau \simeq \tau}$$

$$\text{(CEDL/R)} \quad \frac{}{S \models ? \simeq \tau} \quad \frac{}{S \models \tau \simeq ?}$$

$$\text{(CEF{\small UN})} \quad \frac{S \models \tau_1 \simeq \tau_3 \quad S \models \tau_2 \simeq \tau_4}{S \models \tau_1 \to \tau_2 \simeq \tau_3 \to \tau_4}$$

$$\text{(CEVL/R)} \quad \frac{S \models \tau \sqsubseteq S(\alpha)}{S \models \alpha \simeq \tau} \quad \frac{S \models \tau \sqsubseteq S(\alpha)}{S \models \tau \simeq \alpha}$$

$$\text{(CLV{\small AR})} \quad \frac{S(\alpha) = \tau}{S \models \alpha \sqsubseteq \tau} \qquad \boxed{S \models \tau \sqsubseteq \tau}$$

$$\text{(CLG)} \quad \frac{}{S \models \gamma \sqsubseteq \gamma}$$

$$\text{(CLDL)} \quad \frac{}{S \models ? \sqsubseteq \tau}$$

$$\text{(CLF{\small UN})} \quad \frac{S \models \tau_1 \sqsubseteq \tau_3 \quad S \models \tau_2 \sqsubseteq \tau_4}{S \models \tau_1 \to \tau_2 \sqsubseteq \tau_3 \to \tau_4}$$

Figure 5: The consistent-equal and consistent-less judgments.

using rule (CEVL). There are many choices for α, but whichever choice is made restricts the choices for β. Suppose

$$\{\alpha \mapsto (? \to \mathsf{bool}) \to (\mathsf{int} \to \mathsf{bool})\} \subseteq S$$

Then we have

$$S \models \beta \to (\mathsf{int} \to ?) \sqsubseteq (? \to \mathsf{bool}) \to (\mathsf{int} \to \mathsf{bool})$$

and applying the inverse of (CLFUN) yields

$$S \models \beta \sqsubseteq ? \to \mathsf{bool}, \;\; S \models \mathsf{int} \to ? \sqsubseteq \mathsf{int} \to \mathsf{bool}$$

The second constraint is satisfied by any substitution using (CLFUN), (CLG), and (CLDL), but the first constraint is only satisfied when

$$S(\beta) = (? \to \mathsf{bool})$$

according to rule (CLVAR).

A key property of the $\cdot \models \cdot \simeq \cdot$ judgment is that it allows the two types to differ with respect to ?, but if both sides are variables, then their solutions must be equal, i.e., if $S \models \alpha \simeq \beta$ then $S(\alpha) = S(\beta)$. This is why $\{\alpha \mapsto \mathsf{int}\}$ is a solution for the following program but $\{\alpha \mapsto ?\}$ is not.

```
let f(x:α) =
  let y:β = x in y + 1
```

Proposition 1. *(Properties of $S \models \tau \simeq \tau$ and $S \models \tau \sqsubseteq \tau$)*

1. *$S \models \tau_1 \sqsubseteq \tau_2$ and $S \models \tau_3 \sqsubseteq \tau_2$ implies $S \models \tau_1 \simeq \tau_3$.*
2. *Suppose τ_1 and τ_3 do not contain ?s. Then $S \models \tau_1 \sqsubseteq \tau_2$ and $S \models \tau_1 \simeq \tau_3$ implies $S \models \tau_3 \sqsubseteq \tau_2$.*

3. *If τ_1 and τ_2 contain no ?s and $S \models \tau_1 \simeq \tau_2$, $S(\tau_1) = S(\tau_2)$.*
4. *If τ_1 contains no ?s and $S \models \tau_1 \sqsubseteq \tau_2$, $S(\tau_1) = \tau_2$.*
5. *If $S \models \tau_1 \simeq \tau_2 \to \beta$, then either $\tau_1 = ?$ or there exist τ_{11} and τ_{12} such that $\tau_1 = \tau_{11} \to \tau_{12}$, $\tau_{11} \sim S(\tau_2)$, and $\tau_{12} \sqsubseteq S(\beta)$.*
6. *If $\mathrm{FTV}(\tau_1) = \emptyset$ and $\mathrm{FTV}(\tau_2) = \emptyset$, $S \models \tau_1 \simeq \tau_2$ iff $\tau_1 \sim \tau_2$.*
7. *If $\mathrm{FTV}(\tau_1) = \emptyset$, then $S \models \tau_1 \sqsubseteq \tau_2$ iff $\tau_1 \sqsubseteq \tau_2$.*

4.2 The Definition of the Type System

We formalize our new type system in the setting of the gradually typed lambda calculus with the addition of type variables ($\lambda^{?\alpha}_{\to}$). As in $\lambda^{?}_{\to}$, a parameter that is not annotated is implicitly annotated with the dynamic type. This favors programs that are mostly dynamic. When a program is mostly static, it would be beneficial to instead interpret variables without annotations as being annotated with unique type variables. This option can easily be offered as a command-line compiler flag.

With the consistent-equal judgment in hand we are ready to define the type system for $\lambda^{?\alpha}_{\to}$ with the judgment $S;\Gamma \vdash_g e : \tau$, shown in Figure 6. The crux of the type system is the application rule (GApp). We considered a couple of alternatives before arriving at this rule. First we tried to borrow the (SApp) rule of $\lambda^{\alpha}_{\to}$ (Figure 4) but replace $S(\tau_1) = S(\tau_2 \to \tau_3)$ with $S \models \tau_1 \simeq \tau_2 \to \tau_3$:

$$\frac{S;\Gamma \vdash_g e_1 : \tau_1 \qquad S;\Gamma \vdash_g e_2 : \tau_2 \qquad S \models \tau_1 \simeq \tau_2 \to \tau_3}{S;\Gamma \vdash_g e_1\ e_2 : \tau_3}$$

This rule is too lenient: τ_3 may be instantiated with ? which allows too many programs to type check. Consider the following program.

$$\lambda f : \mathsf{int} \to \mathsf{int}.\ \lambda g : \mathsf{int} \to \mathsf{bool}.\ f\ (g\ 1)$$

The following is a derivation for this program. The problem is that the application $(g\ 1)$ can be given the type ? because $\{\} \models \mathsf{int} \to \mathsf{bool} \simeq \mathsf{int} \to ?$. Let Γ_0 and Γ_1 be the environments defined as follows.

$$\begin{aligned}\Gamma_0 &= \{f : \mathsf{int} \to \mathsf{int}\}\\ \Gamma_1 &= \Gamma_0(g \mapsto (\mathsf{int} \to \mathsf{bool}))\end{aligned}$$

Then we have

$$\dfrac{\dfrac{\dfrac{\{\};\Gamma_1 \vdash_g f : \mathsf{int} \to \mathsf{int} \qquad \dfrac{\begin{array}{c}\{\};\Gamma_1 \vdash_g g : \mathsf{int} \to \mathsf{bool}\\ \{\};\Gamma_1 \vdash_g 1 : \mathsf{int}\end{array}}{\{\};\Gamma_1 \vdash_g g\ 1 : ?}}{\{\};\Gamma_1 \vdash_g (f\ (g\ 1)) : \mathsf{int}}}{\{\};\Gamma_0 \vdash_g (\lambda g : \mathsf{int} \to \mathsf{bool}.\ f\ (g\ 1)) : \mathsf{int}}}{\{\};\vdash_g (\lambda f : \mathsf{int} \to \mathsf{int}.\ \lambda g : \mathsf{int} \to \mathsf{bool}.\ f\ (g\ 1)) : \mathsf{int}}$$

The second alternative we explored was to borrow the (App1) and (App2) rules from $\lambda^{?}_{\to}$, replacing $\tau_1 \sim \tau_2$ with $S \models \tau_1 \simeq \tau_2$.

$$\frac{S;\Gamma \vdash_g e_1 : ? \qquad S;\Gamma \vdash_g e_2 : \tau}{S;\Gamma \vdash_g e_1\ e_2 : ?}$$

$$\frac{S;\Gamma \vdash_g e_1 : \tau_1 \to \tau_3 \qquad S;\Gamma \vdash_g e_2 : \tau_2 \qquad S \models \tau_1 \simeq \tau_2}{S;\Gamma \vdash_g e_1\ e_2 : \tau_3}$$

$$\boxed{S;\Gamma \vdash_g e : \tau}$$

$$(\textsc{GVar}) \qquad \frac{\Gamma(x) = \tau_1}{S;\Gamma \vdash_g x : \tau_1}$$

$$(\textsc{GCnst}) \qquad S;\Gamma \vdash_g c : \mathit{typeof}(c)$$

$$(\textsc{GApp}) \qquad \frac{\begin{array}{c} S;\Gamma \vdash_g e_1 : \tau_1 \qquad S;\Gamma \vdash_g e_2 : \tau_2 \\ S \models \tau_1 \simeq \tau_2 \to \beta \qquad (\beta \text{ fresh}) \end{array}}{S;\Gamma \vdash_g e_1\ e_2 : \beta}$$

$$(\textsc{GAbs}) \qquad \frac{S;\Gamma(x \mapsto \tau_1) \vdash_g e : \tau_2}{S;\Gamma \vdash_g \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}$$

Figure 6: The type system for $\lambda^{?\alpha}_{\to}$.

This alternative also accepts too many programs. Consider the following erroneous program: $((\lambda x : \alpha.\ (x\ 1))\ 1)$. With the substitution $\{\alpha \mapsto ?\}$ this program is well-typed using the first application rule for both applications.

The problem with both of the above approaches is that they allow the type of an application to be ?, thereby adding an extra ? that was not originally in the program. We can overcome this problem by leveraging the definition of the $\simeq$ judgment, particularly with respect to how it treats type variables: it does not allow the solution for a variable to contain more ?s than the types that constrain it. With this intuition we define the (GApp) rule as follows.

$$(\textsc{GApp}) \qquad \frac{\begin{array}{c} S;\Gamma \vdash_g e_1 : \tau_1 \qquad S;\Gamma \vdash_g e_2 : \tau_2 \\ S \models \tau_1 \simeq \tau_2 \to \beta \qquad (\beta \text{ fresh}) \end{array}}{S;\Gamma \vdash_g e_1\ e_2 : \beta}$$

The type of the application is expressed using a type variable instead of a metavariable. This subtle change places a more strict requirement on the variable.

Let us revisit the previous examples and show how this rule correctly rejects them. For the first example

$$\lambda f : \mathsf{int} \to \mathsf{int}.\ \lambda g : \mathsf{int} \to \mathsf{bool}.\ f\ (g\ 1)$$

we have the constraint set

$$\{\mathsf{int} \to \mathsf{bool} \simeq \mathsf{int} \to \beta_1,\ \ \mathsf{int} \to \mathsf{int} \simeq \beta_1 \to \beta_2\}$$

which does not have a solution because β_1 must be the upper bound of int and bool but there is no such upper bound. The second example, $((\lambda x : \alpha.\ (x\ 1))\ 1)$, gives rise to the following set of constraints

$$\{\alpha \simeq \mathsf{int} \to \beta_1,\ \alpha \to \beta_1 \simeq \mathsf{int} \to \beta_2\}$$

which does not have a solution because α would have to be the upper bound of $\mathsf{int} \to \beta_1$ and int.

4.3 Properties of the Type System for $\lambda^{?\alpha}_{\to}$

When there are no type variable annotations in the program, the type system for $\lambda^{?\alpha}_{\to}$ is sound with respect to $\lambda^{?}_{\to}$.

Theorem 2. *Suppose* $\mathrm{FTV}(\Gamma) = \emptyset$ *and* $\mathrm{FTV}(e) = \emptyset$. *If* $S;\Gamma \vdash_g e : \tau$, *then* $\exists \tau'.\ \Gamma \vdash_g e : \tau'$ *and* $\tau' \sqsubseteq S(\tau)$.

Proof. The proof is by induction on the typing derivations. □

The type system for $\lambda^{?\alpha}_{\rightarrow}$ is stronger (accepts strictly fewer programs) than the alternative type system that says there must be a substitution S that makes the program well-typed in $\lambda^{?}_{\rightarrow}$ (Figure 1).

Theorem 3.

1. *If $S; \Gamma \vdash_g e : \tau$ then there is a τ' such that $S(\Gamma) \vdash_g S(e) : \tau'$ and $\tau' \sqsubseteq S(\tau)$.*
2. *If $S(\Gamma) \vdash_g S(e) : \tau$ then it is not always the case that there is a τ' such that $S; \Gamma \vdash_g e : \tau'$.*

Proof.

1. The proof is by induction on the derivation of $S; \Gamma \vdash_g e : \tau$. The case for (GApp) uses Proposition 1, items 2 and 5.
2. Here is a counter example: $(\lambda x : \alpha.\ x\ 1)\ 1$.

□

When there are no ?s in the program, a well-typed $\lambda^{?\alpha}_{\rightarrow}$ program is also well-typed in the completely static type system of $\lambda^{\alpha}_{\rightarrow}$. The contrapositive of this statement says that $\lambda^{?\alpha}_{\rightarrow}$ catches all the type errors that are caught by $\lambda^{\alpha}_{\rightarrow}$.

Theorem 4. *If $e \in \lambda^{\alpha}_{\rightarrow}$ and $(\forall \alpha.\ \Gamma(\alpha) = \tau \Longrightarrow \tau \in \lambda^{\alpha}_{\rightarrow})$ then $S; \Gamma \vdash_g e : \tau$ implies $S; \Gamma \vdash e : \tau$ and $\tau \in \lambda^{\alpha}_{\rightarrow}$.*

Proof. The proof is by induction on the derivation of $S; \Gamma \vdash_g e : \tau$. The case for (GApp) uses Proposition 1 item 3. □

5 A Type Inference Algorithm for $\lambda^{?\alpha}_{\rightarrow}$

The inference algorithm we develop for $\lambda^{?\alpha}_{\rightarrow}$ follows a similar outline to that of the algorithm for $\lambda^{\alpha}_{\rightarrow}$ we presented in Section 2. We generate a set of constraints from the program and then solve the set of constraints. The main difference is that we generate $\simeq$ constraints instead of type equalities, which requires changes to the constraint solver (the unification algorithm).

The classic unification algorithm is not suitable for solving $\simeq$ constraints. Suppose we have the constraint $\{\alpha \rightarrow \alpha \simeq\ ? \rightarrow \mathsf{int}\}$. The unification algorithm would first unify α and ? and substitute ? for α on the other side of the $\rightarrow$. But ? is not a valid solution for α according to the consistent-equal relation: it is not the case that $\mathsf{int} \sqsubseteq\ ?$. The problem with the classic unification algorithm is that it treats the first thing that unifies with a variable as the final solution and eagerly applies substitution. To satisfy the $\simeq$ relation, the solution for a variable must be an upper bound of *all* the types that unify with the variable.

The main idea of our new algorithm is that for each type variable α we maintain a type τ that is a lower bound on the solution of α (i.e. $\tau \sqsubseteq \alpha$). (In contrast, inference algorithms for subtyping maintain both lower and upper bounds [26].) When we encounter another constraint $\alpha \simeq \tau'$, we move the lower bound up to be the least upper bound of τ and τ'. This idea can be integrated with some care into a unification algorithm that does not rely on substitution. The algorithm we present is a variant of Huet's almost linear algorithm [13, 15]. We could have adapted Paterson and Wegman's linear algorithm [21] at the expense of a more detailed and less clear presentation.

$$\text{(CVar)} \quad \frac{\Gamma(x) = \tau_1}{\Gamma \vdash_g x : \tau_1 \mid \{\}} \qquad \boxed{\Gamma \vdash_g e : \tau \mid C}$$

$$\text{(CCnst)} \quad \Gamma \vdash_g c : \mathit{typeof}(c) \mid \{\}$$

$$\text{(CApp)} \quad \frac{\begin{array}{c}\Gamma \vdash_g e_1 : \tau_1 \mid C_1 \\ \Gamma \vdash_g e_2 : \tau_2 \mid C_2 \\ C_3 = \{\tau_1 \simeq \tau_2 \to \beta\} \cup C_1 \cup C_2\end{array}}{\begin{array}{c}\Gamma \vdash_g e_1\ e_2 : \beta \mid C_3 \\ (\beta \text{ fresh})\end{array}}$$

$$\text{(CAbs)} \quad \frac{\Gamma(x \mapsto \tau) \vdash_g e : \rho \mid C}{\Gamma \vdash_g \lambda x : \tau.\ e : \tau \to \rho \mid C}$$

Figure 7: The definition of constraint generation for $\lambda^{?\alpha}_{\to}$.

5.1 Constraint Generation

The constraint generation judgment has the form $\Gamma \vdash_g e : \tau \mid C$, where C is the set of constraints. The constraint generation rules are given in Figure 7 and are straightforward to derive from the type system (Figure 6). The main change is that the side condition on the (GApp) rule becomes a generated constraint on the (CApp) rule. The meaning of a set of these constraints is given by the following definition.

Definition 3. *A set of constraints C is **satisfied** by a substitution S, written $S \models C$, iff for any $\tau_1 \simeq \tau_2 \in C$ we have $S \models \tau_1 \simeq \tau_2$.*

We use one of the previous examples to illustrate constraint generation and, in the next subsection, constraint solving.

$$\lambda f : (? \to \mathsf{int}) \to (\mathsf{int} \to ?) \to \mathsf{int}.\ \lambda y : \alpha.\ f\ y\ y$$

We generate the following constraints from this program.

$$\{(? \to \mathsf{int}) \to (\mathsf{int} \to ?) \to \mathsf{int} \simeq \alpha \to \beta_1, \beta_1 \simeq \alpha \to \beta_2\}$$

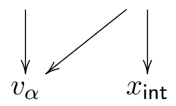
Because of the close connection between the type system and constraint generation, it is straightforward to show that the two are equivalent.

Lemma 3. *Given that $\Gamma \vdash e : \tau \mid C$, $S \models C$ is equivalent to $S; \Gamma \vdash_g e : \tau$.*

Proof. Both directions are proved by induction on the derivation of the constraint generation. □

5.2 Constraint Solver

Huet's algorithm uses a graph representation for types. For example, the type $\alpha \to (\alpha \to \mathsf{int})$ is represented as the node u in the following graph.



Huet used a graph data structure that conveniently combines node labels and out-edges, called the "small term" approach [13, 25]. Each node is labeled with a type, but the type is small in that it consists of either a ground type such as int or a function type ($\rightarrow$) whose parameter and return type are nodes instead of types. For example, the above graph is represented by the following stype function from nodes to shallow types.

$$\begin{array}{ll} \mathsf{stype}(u) = v \rightarrow w & \mathsf{stype}(v) = \mathsf{var} \\ \mathsf{stype}(w) = v \rightarrow x & \mathsf{stype}(x) = \mathsf{int} \end{array}$$

We sometimes write the stype of a node as a subscript, such as $u_{v \rightarrow w}$ and x_{int}. Also, when the identity of a node is not important we sometimes just write the stype label in place of the node (e.g., int instead of x_{int}).

Huet's algorithm uses a union-find data structure [31] to maintain equivalence classes among nodes. The operation $\mathsf{find}(u)$ maps node u to its representative node and performs path compression to speed up later calls to find. The operation $\mathsf{union}(u,v,f)$ merges the classes of u and v. If the argument to f is true then u becomes the representative of the merged class. Otherwise, the representative is chosen based on which class contains more elements, to reduce time complexity.

The definition of our solve algorithm is in Figure 8. We defer discussion of the $\mathsf{copy\_dyn}$ used on the first line. In each iteration of the algorithm we remove a constraint from C, map the pair of nodes x and y to their representatives u and v, and then perform case analysis on the small types of u and v. In each case we merge the equivalence classes for the two nodes and possibly add more constraints. The main difference from Huet's algorithm is some special handling of ?s. When we merge two nodes, we need to decide which one to make the representative and thereby decide which label overrides the other. In Huet's algorithm, a type variable (here nodes labeled var) is overridden by anything else. To handle ?s, we use the rules that ? overrides var but is overridden by anything else. Thus, ? nodes are treated like type variables in that they may merge with any other type. But they are not exactly like type variables in that they override normal type variables. These rules are carried out in cases 3 and 4 of the algorithm.

Before discussing the corner cases of the algorithm ($\mathsf{copy\_dyn}$ and case 2), we apply the algorithm to the running example introduced in Section 5.1. Figure 9 shows a sequence of snapshots of the solver. Snapshot (a) shows the result of converting the generated constraints to a graph. Constraints are represented as undirected double-lines. At each step, we use bold double-lines to indicate the constraints that are about to be eliminated. To get from (a) to (b) we decompose the constraint between the two function types. Nodes that are no longer the representative of their equivalence class are not shown in the graph. Next we process the two constraints on the left, both of which connect a variable to a function type. The function type becomes the representative in both cases, giving us snapshot (c). As before we decompose a constraint between the two function types into constraints on their children and we have snapshot (d). We first merge the variable node for β_2 into the int node to get (e) and then decompose the constraint between the function type nodes into two more constraints in (f). Here we have constraints on nodes labeled with the ? type. In both cases the node labeled int overrides ? and becomes the representative. The final state is shown in snapshot (g), from which the solutions for the type variables can be read off. As expected, we have $\alpha = \mathsf{int} \rightarrow \mathsf{int}$.

Case 2 of the algorithm, for $? \simeq v_1 \rightarrow v_2$, deserves some explanation. Consider the program $(\lambda f : ?.\ \lambda x : \alpha.\ f\ x)$. The set of constraints generated from this is $\{? \simeq \alpha \rightarrow \beta\}$. According to the operational semantics from Siek and Taha [29], f is cast to $? \rightarrow ?$, so in some sense, we really should have the constraint $? \rightarrow ? \simeq \alpha \rightarrow \beta$. To simulate this in the algorithm we insert two constraints: $? \simeq v_1$ and $? \simeq v_2$. Now, some care must be taken to prevent infinite loops. Consider the constraint $? \simeq v$ where $\mathsf{stype}(v) = v \rightarrow v$. The two new constraints are identical

```
solve($C$) =
  $C$ := copy_dyn($C$)
  for each node $u$ do
    $u$.contains_vars := true
  end for
  while not $C$.empty() do
    $x \simeq y$ := $C$.pop()
    $u$ := find($x$); $v$ := find($y$)
    if $u \neq v$ then
      $(u, v, f)$ := order($u$,$v$)
      union($u$, $v$, $f$)
      case stype($u$) $\simeq$ stype($v$) of
         $u_1 \to u_2 \simeq v_1 \to v_2 \Rightarrow$ (* case 1 *)
            $C$.push($u_1, v_1$); $C$.push($u_2, v_2$)
        | $u_1 \to u_2 \simeq ?$ $\Rightarrow$ (* case 2 *)
            if $u$.contains_vars then
              $u$.contains_vars := false
              $w_1$ = new_vertex(stype=?, contains_vars=false)
              $w_2$ = new_vertex(stype=?, contains_vars=false)
              $C$.push($w_1 \simeq u_1$); $C$.push($w_2 \simeq u_2$)
        | $\tau \simeq$ var | $\tau \simeq ?$ $\Rightarrow$ (* pass, case 3 and 4 *)
        | $\gamma \simeq \gamma \Rightarrow$ (* pass, case 5 *)
        | _ $\Rightarrow$ error: inconsistent types (* case 6 *)
  end while
  $G$ = the quotient of the graph by the equivalence classes
  if $G$ is acyclic then
    return $\{u \mapsto$ stype(find($u$)) | $u$ a node in the graph$\}$
  else error

order($u$,$v$) = case stype($u$) $\simeq$ stype($v$) of
                 | $? \simeq \alpha \Rightarrow (u, v,$ true$)$
                 | $? \simeq \tau$ | $\alpha \simeq \tau \Rightarrow (v, u,$ true$)$
                 | $\tau \simeq \alpha$ | $\tau \simeq ?$$\Rightarrow (u, v,$ true$)$
                 | _ $\Rightarrow (u, v,$ false$)$
```

Figure 8: The constraint solving algorithm.

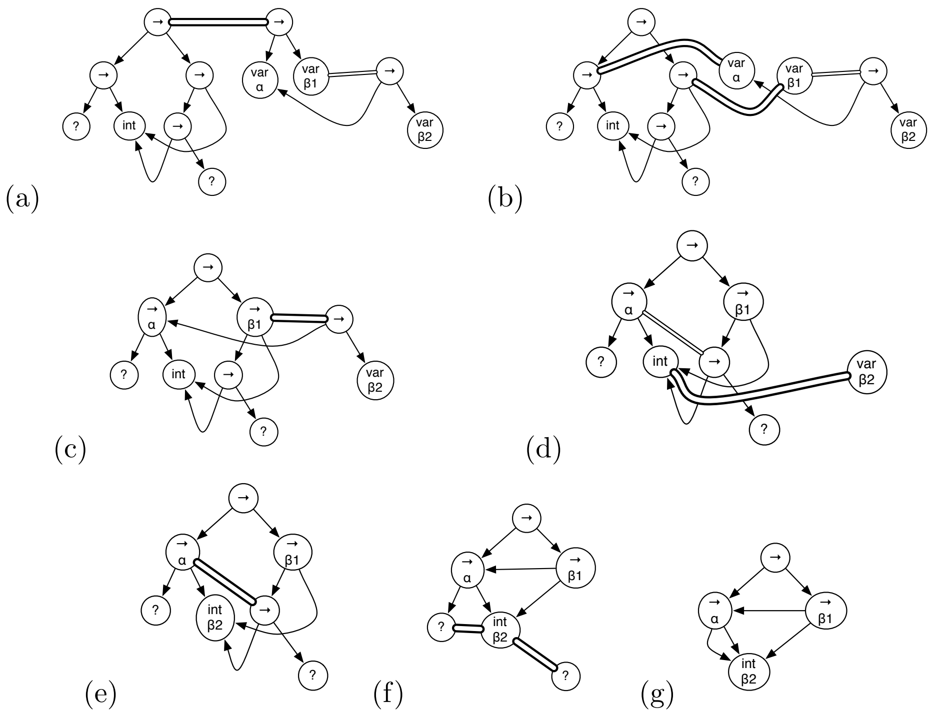


Figure 9: An example run of the constraint solver.

to the original. To avoid this problem we mark each node to indicate whether it may contain a variable. The flags are initialized to true and when we see the constraint $? \simeq v$ we change the flag to false.

The copy_dyn function replaces each node labeled ? with a new node labeled ?, thereby removing any sharing of ? nodes. This is necessary to allow certain programs to type check, such as the example in Section 3 with the functions f, g, and h. The following is a simplified example that illustrates the same problem.

$$\lambda f : \mathsf{int} \to \mathsf{bool} \to \mathsf{int}.\ \lambda x :?.\ f\ x\ x$$

From this program we get the constraint set

$$\{\mathsf{int} \to \mathsf{bool} \to \mathsf{int} \simeq\ u_? \to v,\ v \simeq\ u_? \to w\}$$

If we forgo the copy_dyn conversion and just run the solver, we ultimately get $\mathsf{int} \simeq u_?$ and $\mathsf{bool} \simeq u_?$ which will result in an error. With the copy_dyn conversion, the two occurrences of $u_?$ are replaced by separate nodes that can separately unify with int and bool and avoid the error. It is important that we apply the copy_dyn conversion to the generated constraints and not to the original program, as that would not avoid the above problem.

The infer function, defined in the following, is the overall inference algorithm, combining constraint generation and solving.

Definition 4. *(Inference algorithm) Given* Γ *and* e*, let* τ*,* C*, and* S *be such that* $\Gamma \vdash e : \tau \mid C$ *and* $S = \mathsf{solve}(C)$*. Then* $\mathsf{infer}(\Gamma, e) = (S, S(\tau))$.

5.3 Properties of the inference algorithm

The substitution S returned from the solver is not idempotent. It can be turned into an idempotent substitution by applying it to itself until a fixed point is reached, which we denote by S^*. Note that the solution S' returned by solve is less or equally informative than the other solutions, thereby avoiding types that would introduce unnecessary cast errors.

Lemma 4. *(Soundness and completeness of the solver)*

1. *If* $S = \mathsf{solve}(C)$ *then* $S^* \models C$.
2. *If* $S \models C$ *then* $\exists S'R.\ S' = \mathsf{solve}(C)$ *and* $R \circ S'^* \sqsubseteq S$.

Proof. The correctness of the algorithm is based on the following invariant. Let C be the original set of constraints and C' the set of constraints at a given iteration of the algorithm. At each iteration of the algorithm, $S \models C$ if an only if

1. $S \models C'$,
2. for every pair of type variables α and β in the same equivalence class, $S(\alpha) = S(\beta)$, and
3. there is an R such that $R \circ S' \sqsubseteq S$, where S' is the current solution based on the stype and union-find data structures.

When the algorithm starts, $C = C'$, so the invariant holds trivially. The invariant is proved to hold at each step by case analysis. Once the algorithm terminates, we read off the answer based on the stype and the union-find data structure. This gives a solution that is less informative but more general (in the Hindley-Milner sense) than any other solution, expressed by the clause $R \circ S'^* \sqsubseteq S$. □

Lemma 5. *The time complexity of the* solve *algorithm is* $O(m\alpha(n))$*, where* n *is the number of nodes and* m *is the number of edges.*

Proof. The number of iterations in the solve algorithm is $O(m)$. In case 1 of the algorithm we push two constraints into C and make the v node and its two out-edges inaccessible from the find operation. In case 2 of the algorithm, we push two constraints into C and we mark the function type node as no-longer possibly containing variables, which makes it and its two out-edges inaccessible to subsequent applications of case 2. Each iteration performs union-find operations, which have an amortized cost of $\alpha(n)$ [31], so the overall time complexity is $O(m\alpha(n))$. □

Theorem 5. *(Soundness and completeness of inference)*

1. *If* $(S, \tau) = \mathsf{infer}(\Gamma, e)$*, then* $S^*; \Gamma \vdash_g e : \tau$.
2. *If* $S; \Gamma \vdash_g e : \tau$ *then there is a* S', τ', *and* R *such that* $(S', \tau') = \mathsf{infer}(\Gamma, e)$, $R \circ S'^* \sqsubseteq S$, *and* $R \circ S'^*(\tau') \sqsubseteq S(\tau)$.

Proof. Let τ' and C be such that $\Gamma \vdash e : \tau' | C$.

1. By the soundness of solve (Lemma 4) we have $S^* \models C$. Then by the equivalence of constraint generation and the type system (Lemma 3), we have $S^*; \Gamma \vdash e : \tau$.
2. By the equivalence of constraint generation and the type system (Lemma 3), we have $S \models C$. Then by the completeness of solve (Lemma 4) there exists S' and R such that $S' = \mathsf{solve}(C)$ and $R \circ S'^* \sqsubseteq S$. We then conclude using the definition of infer.

□

Theorem 6. *The time complexity of the* infer *algorithm is* $O(n\alpha(n))$ *where* n *is the size of the program.*

Proof. The constraint generation step is $O(n)$ and the solver is $O(n\alpha(n))$ (the number of edges in the type graph is bounded by $2n$ because no type has out-degree greater than 2) so the overall time complexity is $O(n\alpha(n))$. □

6 Related Work

The interface between dynamic and static typing has been a fertile area of research. We cite a limited number of papers for lack of space. The reader may refer to the references in the cited papers for more detailed lists for each topic.

Optional Types in Dynamic Languages Many dynamic languages allow explicit type annotations. Common LISP [14] is an example. In Common LISP, adding type annotations improves performance but the language does not make the guarantee that annotating all parameters in the program prevents all cast errors at run-time, as is the case for gradual typing. More recently, Tobin-Hochstadt and Felleisen [33, 34] developed a type system for Scheme that facilitates migration between dynamic and static code on a per-module basis.

Type Inference There is a huge body of literature on the topic of type inference, especially regarding variations of the Hindley-Milner type system [17]. Of that, the closest to our work is that on combining inference and subtyping [4, 26]. The main difference between inference for subtyping versus gradual typing is that subtyping has co/contra-variance in function types, whereas the consistency relation is covariant in both the parameter and return type, making the inference problem for gradual typing more tractable.

Gradual Typing In addition to the related work discussed in the introduction, we mention a couple more related works here. Anderson and Drossopoulou developed a gradual type system for BabyJ [2] that uses nominal types. Gronski, Knowles, Tomb, Freund, and Flanagan [9] provide gradual typing in the Sage language by including a Dynamic type and implicit down-casts. They use a modified form of subtyping to provide the implicit down-casts.

Quasi-static Typing Thatte's Quasi-Static Typing [32] is close to gradual typing but relies on subtyping and treats the unknown type as the top of the subtype hierarchy. Siek and Taha [29] show that implicit down-casts combined with the transitivity of subtyping creates a fundamental problem that prevents this type system from catching all type errors even when all parameters in the program are annotated.

Soft Typing Static analyses based on dataflow can be used to perform static checking and to optimize performance. The later variant of Soft Typing by Flanagan and Felleisen [7] is an example of this approach. These analyses provide warnings to the programmer while still allowing the programmer to execute their program immediately (even programs with errors), thereby preserving the benefits of dynamic typing. However, the programmer does not control which portions of a program are statically checked: these whole-program analyses have non-local interactions.

Dynamic Typing in Statically Typed Languages Abadi et al. [1] extended a statically typed language with a Dynamic type and explicit injection (dynamic) and projection operations (typecase). Their approach does not satisfy the goals of gradual typing, as migrating code between dynamic and static checking not only requires changing type annotations on parameters, but also adding or removing injection and projection operations throughout the code. Gradual typing automates the latter.

Hybrid Typing The Hybrid Type Checking of Flanagan [5] combines standard static typing with refinement types, where the refinements may express arbitrary predicates. This is analogous to gradual typing in that it combines a weaker and stronger type system, allowing implicit coercions between the two systems and inserting run-time checks. A notable difference is that hybrid typing is based on subtyping whereas gradual typing is based on type consistency.

7 Conclusion

This paper develops a type system for the gradually typed lambda calculus with type variables ($\lambda^{?\alpha}_{\to}$). The system integrates type inference and gradual typing to aid programmers in adding types to their programs. In the proposed system, a programmer uses a type variable annotation to request the best solution for the variable from the inference algorithm.

The type system presented satisfies the defining properties of a gradual type system. That is, a programmer may omit type annotations on function parameters and immediately run the program; run-time type checks are performed to preserve type safety. Furthermore, a programmer may add type annotations to increase static checking. When all function parameters are annotated, all type errors are caught at compile-time.

The paper also develops an efficient inference algorithm for $\lambda^{?\alpha}_{\to}$ that is sound and complete with respect to the type system and that takes care not to infer types that would introduce cast errors.

A Isabelle Formalization

A.1 Syntax and Auxilliary Functions

```
types name = nat

datatype ty =
    UVarT name
  | IntT          (int)
  | BoolT         (bool)
  | DynT          (?)
  | ArrowT ty ty (infixr → 95)

datatype const = IntC int
               | BoolC bool
               | Succ
               | IsZero

datatype expr =
    Var name
  | Const const
  | Lam name ty expr (λ-:-. - [53,53,53] 52)
  | App expr expr

types env = (name × ty) list
types subst = (name × ty) list

axclass  type-struct < type

instance ty::type-struct ..
instance expr::type-struct ..
instance nat::type-struct ..
instance option::(type-struct)type-struct ..
```

```
instance fun::(type, type-struct)type-struct ..
instance list::(type-struct)type-struct ..
instance set::(type-struct)type-struct ..
instance ∗::(type-struct,type-struct)type-struct ..
```

A.1.1 Auxilliary Functions

```
constdefs id-subst :: subst
  id-subst-def[simp]: id-subst ≡ []

— domain of an association list
consts
  Dom :: ('a × 'b) list ⇒ 'a set
primrec
  Dom [] = {}
  Dom (xt#ls) = insert (fst xt) (Dom ls)

consts
  lookup :: ('a × 'b) list ⇒ 'a ⇒ 'b option
primrec
  lookup [] k = None
  lookup (kv#ls) k =
    (if fst kv = k then Some (snd kv) else lookup ls k)

constdefs
  lookup-subst :: (name × ty) list ⇒ name ⇒ ty (%)
  lookup-subst S a ≡
    (case (lookup S a) of None ⇒ UVarT a | Some t ⇒ t)

consts
  app-subst :: [subst, 'a::type-struct] => 'a::type-struct ($)
syntax (latex)
  app-subst :: [subst, 'a::type-struct] => 'a::type-struct ()

primrec (app-subst-ty)
  $S (UVarT a) = % S a
  $S (IntT) = (IntT)
  $S (BoolT) = (BoolT)
  subst-fun: $S (t1 → t2) = ($S t1) → ($S t2)
  $S (?) = (?)

primrec (app-subst-list)
  $S [] = []
  $S (x#xs) = ($S x)#($S xs)

defs (overloaded)
  app-subst-pair: $S p ≡ (fst p, $S (snd p))

primrec (app-subst-expr)
subst-var:   $S (Var x) = (Var x)
             $S (Const c) = (Const c)
subst-abs:   $S (λx:τ. e) = (λx:$S τ. $S e)
             $S (App e1 e2) = (App ($S e1) ($S e2))

primrec (app-subst-option)
  app-subst S None = None
```

$\textit{app-subst}\ S\ (\textit{Some}\ \tau) = \textit{Some}\ (\textit{app-subst}\ S\ \tau)$

defs (**overloaded**)
$\textit{app-subst-fun}\colon \textit{app-subst}\ S\ \Gamma \equiv (\lambda\ x.\ \textit{app-subst}\ S\ (\Gamma\ x))$

consts $\textit{FTV} :: {}'a{::}\textit{type-struct} \Rightarrow \textit{nat set}$

primrec ($\textit{FTV-ty}$)
$\textit{FTV}\ (\textit{UVarT}\ \alpha) = \{\alpha\}$
$\textit{FTV}\ (\textit{IntT}) = \{\}$
$\textit{FTV}\ (\textit{BoolT}) = \{\}$
$\textit{FTV}\ (\textit{DynT}) = \{\}$
$\textit{FTV}\ (\textit{t1} \to \textit{t2}) = \textit{FTV}\ \textit{t1} \cup \textit{FTV}\ \textit{t2}$

primrec ($\textit{FTV-expr}$)
$\textit{FTV}\ (\textit{Var}\ x) = \{\}$
$\textit{FTV}\ (\textit{Const}\ c) = \{\}$
$\textit{FTV}\ (\lambda\ x{:}\tau.\ e) = \textit{FTV}\ \tau \cup \textit{FTV}\ e$
$\textit{FTV}\ (\textit{App}\ \textit{e1}\ \textit{e2}) = \textit{FTV}\ \textit{e1} \cup \textit{FTV}\ \textit{e2}$

primrec ($\textit{FTV-option}$)
$\textit{FTV None} = \{\}$
$\textit{FTV}\ (\textit{Some}\ \tau) = \textit{FTV}\ \tau$

primrec ($\textit{FTV-list}$)
$\textit{FTV}\ [] = \{\}$
$\textit{FTV}\ (a\#\textit{ls}) = \textit{FTV}\ a \cup \textit{FTV}\ \textit{ls}$

defs (**overloaded**)
$\textit{FTV-nat}[\textit{simp}]\colon \textit{FTV}\ x \equiv \{x\}$

defs (**overloaded**)
$\textit{FTV-pair}[\textit{simp}]\colon \textit{FTV}\ p \equiv \textit{FTV}\ (\textit{snd}\ p)$

defs (**overloaded**)
$\textit{FTV-set}\colon \textit{FTV}\ C \equiv \{\alpha\ .\ \exists\ e.\ \alpha \in \textit{FTV}\ e \wedge e \in C\ \}$

defs (**overloaded**)
$\textit{FTV-fun}\colon \textit{FTV}\ \Gamma \equiv \{\alpha.\ \exists\ y\ t.\ \Gamma\ y = t \wedge \alpha \in \textit{FTV}\ t\}$

consts $\textit{no-dyn} :: {}'a{::}\textit{type-struct} \Rightarrow \textit{bool}$

primrec ($\textit{no-dyn-ty}$)
$\textit{no-dyn}\ (\textit{UVarT}\ \alpha) = \textit{True}$
$\textit{no-dyn}\ (\textit{IntT}) = \textit{True}$
$\textit{no-dyn}\ (\textit{BoolT}) = \textit{True}$
$\textit{no-dyn}\ (\textit{DynT}) = \textit{False}$
$\textit{no-dyn}\ (\textit{t1} \to \textit{t2}) = (\textit{if}\ (\textit{no-dyn}\ \textit{t1})\ \textit{then}\ (\textit{no-dyn}\ \textit{t2})\ \textit{else False})$

primrec ($\textit{no-dyn-expr}$)
$\textit{no-dyn}\ (\textit{Var}\ x) = \textit{True}$
$\textit{no-dyn}\ (\textit{Const}\ c) = \textit{True}$
$\textit{no-dyn}\ (\lambda x{:}\tau.\ e) = (\textit{if no-dyn}\ \tau\ \textit{then no-dyn}\ e\ \textit{else False})$
$\textit{no-dyn}\ (\textit{App}\ \textit{e1}\ \textit{e2}) = (\textit{if no-dyn}\ \textit{e1}\ \textit{then no-dyn}\ \textit{e2}\ \textit{else False})$

```
defs (overloaded)
  no-dyn-fun: no-dyn Γ ≡ (∀ x a. Γ x = a ⟶ no-dyn a)

primrec (no-dyn-option)
  no-dyn None = True
  no-dyn (Some τ) = no-dyn τ

defs (overloaded)
  no-dyn-pair[simp]: no-dyn p ≡ no-dyn (snd p)

primrec (no-dyn-list)
  no-dyn [] = True
  no-dyn (a#ls) = (if no-dyn a then no-dyn ls else False)

constdefs idempotent :: subst ⇒ bool
  idempotent S ≡ $S (% S) = % S
```

A.1.2 Properties of Auxilliary Functions

```
lemma finite-ftv-ty[intro!]: finite (FTV (τ::ty))
  apply (induct τ) by auto

lemma finite-ftv-expr[intro!]: finite (FTV (e::expr))
  apply (induct e) by auto

lemma finite-ftv-subst[intro!]: finite (FTV (S::subst))
  apply (induct S) by auto

lemma id-id[simp]: $id-subst (τ::ty) = τ apply (induct τ)
  using lookup-subst-def by auto

lemma closed-subst-id: FTV τ = {} ⟹ $S τ = (τ::ty)
  apply (induct τ) by auto

lemma idempotent-ty[rule-format]:
  ∀ S. idempotent S ⟶ $ S ($ S t) = $ S (t::ty)
  apply (induct t) defer apply simp apply simp apply simp apply simp
proof −
  fix n::nat
  show ∀ S. idempotent S ⟶ $ S ($ S (UVarT n)) = $ S (UVarT n)
  proof clarify
    fix S assume id: idempotent S
    have $ S ($ S (UVarT n)) = $ S (%S n) by simp
    also have ... = ($S (%S)) n by (simp add: app-subst-fun)
    also from id have ... = %S n by (simp add: idempotent-def)
    also have ... = $S (UVarT n) by simp
    finally show $ S ($ S (UVarT n)) = $ S (UVarT n) by blast
  qed
qed

lemma ftv-dom-id[rule-format]:
  ∀ S. (∀ a. a ∈ FTV τ ⟶ % S a = UVarT a) = ($S τ = (τ::ty))
  apply (induct τ) by auto

— This is Lemma 2 of the paper
lemma t1tot2eqSt-implies-t2eqSt2[rule-format]:
```

$\mathit{idempotent}\ S \wedge (\$S\ \tau = \tau_1 \to \tau_2) \longrightarrow \tau_2 = \$S\ \tau_2$
apply (*induct-tac* τ) **defer apply** *force* **apply** *force* **apply** *force*
apply *simp* **apply** (*rule impI*) **defer apply** (*rule impI*)
proof –
fix *a*
assume *tmp*: $\mathit{idempotent}\ S \wedge \$S\ (\mathit{UVarT}\ a) = \tau_1 \to \tau_2$
from *tmp* **have** *idems*: *idempotent S* **by** *simp*
from *idems* **have** *sstt*: $\$S\ (\$S\ (\mathit{UVarT}\ a)) = (\$S\ (\mathit{UVarT}\ a))$
by (*rule idempotent-ty*)
with *tmp sstt* **have** *sneqst1tost2*: $(\$S\ (\mathit{UVarT}\ a)) = \$S\ \tau_1 \to \$S\ \tau_2$ **by** *simp*
with *tmp* **show** $\tau_2 = \$S\ \tau_2$ **by** *simp*
next
fix *ty1 ty2*
assume *styc*: $\mathit{idempotent}\ S \wedge \$S\ \mathit{ty1} = \tau_1 \wedge \$S\ \mathit{ty2} = \tau_2$
hence *idempotent S* **by** *simp*
hence $\$S\ (\$S\ \mathit{ty2}) = \$S\ \mathit{ty2}$ **by** (*rule idempotent-ty*)
with *styc* **show** $\tau_2 = \$S\ \tau_2$ **by** *simp*
qed

lemma *Steqt1tot2-implies-t2eqSt2*[*rule-format*]:
$\mathit{idempotent}\ S \wedge (\tau_1 \to \tau_2 = \$S\ \tau) \longrightarrow \tau_2 = \$S\ \tau_2$
proof –
have $\mathit{idempotent}\ S \wedge \$S\ \tau = \tau_1 \to \tau_2 \longrightarrow \tau_2 = \$S\ \tau_2$
using *t1tot2eqSt-implies-t2eqSt2* **by** *blast*
thus $\mathit{idempotent}\ S \wedge (\tau_1 \to \tau_2 = \$S\ \tau) \longrightarrow \tau_2 = \$S\ \tau_2$ **by** *auto*
qed

lemma *Steqt1tot2-implies-st2eqt2*:
$\llbracket\ \mathit{idempotent}\ S;\ \tau_1 \to \tau_2 = \$S\ \tau\ \rrbracket \Longrightarrow \$S\ \tau_2 = \tau_2$
using *Steqt1tot2-implies-t2eqSt2* **by** *auto*

A.2 The Simply Typed Lambda Calculus

consts *TypeOf* :: *const* ⇒ *ty*
primrec
TypeOf (*IntC n*) = *IntT*
TypeOf (*BoolC b*) = *BoolT*
TypeOf Succ = *IntT* → *IntT*
TypeOf IsZero = *IntT* → *BoolT*

inductive *stlc-wt* :: *env* ⇒ *expr* ⇒ *ty* ⇒ *bool* (- ⊢ - : - [*52*,*52*,*52*] *51*)
where
Var[*intro*!]: $\llbracket\ \mathit{lookup}\ \Gamma\ x = \mathit{Some}\ \tau\ \rrbracket \Longrightarrow \Gamma \vdash \mathit{Var}\ x : \tau$ |
Const[*intro*!]: $\Gamma \vdash \mathit{Const}\ c : \mathit{TypeOf}\ c$ |
Abs[*intro*!]: $\llbracket\ (x,\tau_1)\#\Gamma \vdash e : \tau_2\ \rrbracket \Longrightarrow \Gamma \vdash (\lambda x{:}\tau_1.\ e) : \tau_1 \to \tau_2$ |
App[*intro*!]: $\llbracket\ \Gamma \vdash e : \tau_1 \to \tau_2;\ \Gamma \vdash e' : \tau_1\ \rrbracket$
$\Longrightarrow \Gamma \vdash (\mathit{App}\ e\ e') : \tau_2$
inductive *istlc-wt* :: [*subst*,*env*] ⇒ [*expr*,*ty*] ⇒ *bool* (-;- ⊢ - : - [*52*,*52*,*52*,*52*] *51*)
where
SVar[*intro*!]: $\llbracket\ \mathit{lookup}\ \Gamma\ x = \mathit{Some}\ \tau\ \rrbracket \Longrightarrow S;\Gamma \vdash \mathit{Var}\ x : \tau$ |

SConst[*intro*!]: $\tau = \mathit{TypeOf}\ c \Longrightarrow S;\Gamma \vdash \mathit{Const}\ c : \tau$ |

SAbs[*intro*!]: $\llbracket\ S;(x,\tau_1)\#\Gamma \vdash e : \tau_2\ \rrbracket \Longrightarrow S;\Gamma \vdash (\lambda x{:}\tau_1.\ e) : \tau_1 \to \tau_2$ |

```
  SApp[intro!]: ⟦ S;Γ ⊢ e : τ1; S;Γ ⊢ e' : τ2; $S τ1 = $S (τ2 → τ3) ⟧
           ⟹ S;Γ ⊢ (App e e') : τ3

lemma ex-t[rule-format]: ∀ S x. lookup ($(S::subst) Γ'::env) x = Some τ ⟶
  (∃ τ'. lookup Γ' x = Some τ' ∧ $S τ' = τ)
  apply (induct Γ')
  apply simp
  apply clarify apply (simp add: app-subst-pair)
    apply (case-tac a = x) apply simp
    apply auto
    done

lemma idem-ftvst-impl:
  ∀ S a. idempotent S ∧ a ∈ FTV ($S (τ::ty)) ⟶ %S a = UVarT a
  apply (induct τ)
  defer apply simp apply simp apply simp apply simp apply blast
  apply clarify
proof −
  fix b S a
  assume ids: idempotent S and aftv: a ∈ FTV ($ S (UVarT b))
  from ids have $S ($S (UVarT b)) = $S (UVarT b) by (rule idempotent-ty)
  hence ∀ a. a ∈ FTV ($ S (UVarT b)) ⟶ %S a = UVarT a using ftv-dom-id by blast
  with aftv show % S a = UVarT a by simp
qed

lemma idem-ftvst:
  ⟦ idempotent S; a ∈ FTV ($S (τ::ty)) ⟧ ⟹ %S a = UVarT a
  using idem-ftvst-impl by blast

lemma ftv-wt-sub-impl: Γ' ⊢ e' : τ ⟹
   ∀ Γ e S. idempotent S ∧ Γ' = $S Γ ∧ e' = $S e
   ⟶ (∀ a. a ∈ FTV τ ⟶ % S a = UVarT a)
  apply (induct rule: stlc-wt.induct)
  defer
  apply (case-tac c) apply force apply force apply force apply force
  apply clarify apply (case-tac ea) apply force apply force prefer 2 apply force
    apply simp apply (erule-tac x=(x,τ1)#Γ' in allE)
      apply (erule-tac x=expr in allE)
      apply (erule-tac x=S in allE) apply (erule impE) apply simp
      apply (simp add: app-subst-pair) apply (simp add: idempotent-ty)
      apply clarify  apply (erule disjE) apply simp
        using idem-ftvst apply simp apply simp
  apply clarify apply (case-tac ea) apply force apply force apply force
    apply (erule-tac x=Γ' in allE) apply (erule-tac x=Γ in allE)
    apply (erule-tac x=expr1 in allE) apply (erule-tac x=expr2 in allE)
    apply (erule-tac x=S in allE) apply (erule-tac x=S in allE)
    apply (erule impE) apply (simp add: app-subst-fun)
    apply simp
  apply clarify
proof −
  fix Γ x and τ::ty and Γ' e and S::subst and a
  assume sgx: lookup ($ S Γ') x = Some τ and ids: idempotent S
    and xse: Var x = $ S e and aft: a ∈ FTV τ
  from sgx have X: ∃ τ'. lookup Γ' x = Some τ' ∧ $S τ' = τ by (rule ex-t)
  from X obtain τ' where gpx: lookup Γ' x = Some τ' and stp: $S τ' = τ by blast
```

```
  from aft stp have afst: a ∈ FTV ($S τ') by simp
  from ids afst show % S a = UVarT a by (rule idem-ftvst)
qed

lemma ftv-wt-sub: ⟦ $S Γ ⊢ $S e : τ; idempotent S ⟧
  ⟹ (∀ a. a ∈ FTV τ ⟶ %S a = UVarT a)
  using ftv-wt-sub-impl apply blast done

— Lemma 1 of the paper
lemma ewt-steqt:
  assumes idems: idempotent S and ewt: $S Γ ⊢ $S e : τ
  shows  $S τ = τ
  using idems ewt ftv-wt-sub ftv-dom-id by blast

lemma ewt-ewSt:
  assumes idems: idempotent S and ewt: $S Γ ⊢ $S e : τ
  shows $S Γ ⊢ $S e : $S τ
proof −
    from ewt idems have (∀ a. a ∈ FTV τ ⟶ %S a = UVarT a) by (rule ftv-wt-sub)
    hence $S τ = τ using ftv-dom-id by blast
    thus $S Γ ⊢ $S e : $S τ by simp
qed

lemma ewt-teqSt:
  assumes idems: idempotent S and ewt: $S Γ ⊢ $S e : τ
  shows  τ = $S τ
proof −
  from idems ewt have $S τ = τ by (rule ewt-steqt)
  thus τ = $S τ by auto
qed

lemma stlc-implies-istlc-impl:
  Γ' ⊢ e' : τ' ⟹
    (∀ Γ e S τ . idempotent S ∧ Γ' = $S Γ ∧ e' = $S e ∧ τ' = ($S τ)
  ⟶ (∃ τ'' . (S;Γ ⊢ e : τ'' ∧ $S τ'' = τ')))
  apply (induct rule: stlc-wt.induct)
  apply clarify defer apply clarify defer apply clarify defer apply clarify defer
proof −
  fix Γ x τ and Γ'::env and e::expr and S::subst and τ'
  assume sgx: lookup ($ S Γ') x = Some ($ S τ') and ids: idempotent S
    and vxe: Var x = $ S e
  from sgx ex-t obtain τ'' where lgx: lookup Γ' x = Some τ''
    and stst: $S τ'' = $S τ' by blast
  from vxe lgx stst show ∃τ''. S;Γ' ⊢ e : τ'' ∧ $ S τ'' = $ S τ'
    apply (case-tac e::expr) apply (rule-tac x=τ'' in exI) by auto
next
  fix Γ c Γ' e S τ
  assume idempotent S and Const c = $ S e and TypeOf c = $ S τ
  thus ∃τ''. S;Γ' ⊢ e : τ'' ∧ $ S τ'' = TypeOf c
    apply (rule-tac x=TypeOf c  in exI)
    apply (simp add: idempotent-ty)
    apply (case-tac e::expr) apply auto done
next
  fix x τ₁ Γ e τ₂ Γ' ea S τ
  assume IH1: ∀Γ ea Sa τ.
```

```
          idempotent Sa ∧
          (x, τ1) # $ S Γ′ = $ Sa Γ ∧ e = $ Sa ea ∧ τ2 = $ Sa τ ⟶
          (∃τ″. Sa;Γ ⊢ ea : τ″ ∧ $ Sa τ″ = τ2)
    and ids: idempotent S and le: λx:τ1. e = $ S ea and t12st: τ1 → τ2 = $ S τ
  from le obtain t b where ea: ea = λx:t. b and t1st: τ1 = $S t
    and esb: e = $S b apply (case-tac ea::expr) apply auto done
  from ids t12st have t2st2: τ2 = $S τ2 using Steqt1tot2-implies-t2eqSt2 by blast
  from ids IH1 t1st ea esb
  have X: ∃ τ″. S;(x,t)#Γ′ ⊢ b : τ″ ∧ $S τ″ = τ2
    apply auto apply (erule-tac x=(x,t)#Γ′ in allE)
       apply (erule-tac x=b in allE)
       apply (erule-tac x=S in allE)
       apply (erule-tac x=τ2 in allE) apply auto
       apply (simp add: app-subst-pair) using t2st2 apply simp done
  from X obtain t2 where wtb: S;(x,t)#Γ′ ⊢ b : t2 and st2t2: $S t2 = τ2 by blast
  from wtb have wtl: S;Γ′ ⊢ λx:t. b : t → t2 by blast
  with ea st2t2 t1st
  show ∃τ″. S;Γ′ ⊢ ea : τ″ ∧ $ S τ″ = τ1 → τ2 by auto
next
  fix Γ e τ1 τ2 e′ Γ′ ea S τ
  assume wte: $ S Γ′ ⊢ e : τ1 → $ S τ
    and IH1: ∀Γ ea Sa τ′.
          idempotent Sa ∧
          $ S Γ′ = $ Sa Γ ∧ e = $ Sa ea ∧ τ1 → $ S τ = $ Sa τ′ ⟶
          (∃τ″. Sa;Γ ⊢ ea : τ″ ∧ $ Sa τ″ = τ1 → $ S τ)
    and wtep: $ S Γ′ ⊢ e′ : τ1
    and IH2: ∀Γ e Sa τ.
          idempotent Sa ∧ $ S Γ′ = $ Sa Γ ∧ e′ = $ Sa e ∧ τ1 = $ Sa τ ⟶
          (∃τ″. Sa;Γ ⊢ e : τ″ ∧ $ Sa τ″ = τ1)
    and ids: idempotent S and A: App e e′ = $ S ea
  from A obtain e1 e2 where EA: ea = App e1 e2 and E: e = $S e1
    and EP: e′ = $S e2 apply (case-tac ea::expr) by auto
  from ids wte E have τ1 → $ S τ = $S (τ1 → $ S τ) using ewt-teqSt by blast
  hence t1st1: τ1 = $S τ1 by simp
  from ids E IH1 t1st1 obtain t1 where wte1: S;Γ′ ⊢ e1 : t1
    and st1t1st: $S t1 = τ1 → $ S τ
    apply auto apply (erule-tac x=Γ′ in allE)
    apply (erule-tac x=e1 in allE)
    apply (erule-tac x=S in allE)
    apply (erule-tac x=τ1 → $ S τ in allE)
    apply auto using idempotent-ty apply simp done
  from ids EP IH2 obtain t2 where wte2: S;Γ′ ⊢ e2 : t2 and st2t1: $ S t2 = τ1
    apply simp apply (erule-tac x=Γ′ in allE)
    apply (erule-tac x=e2 in allE)
    apply (erule-tac x=S in allE)
    apply (erule-tac x=τ1 in allE)
    apply auto using t1st1 apply simp done
  from st1t1st st2t1 have eq: $S t1 = $S (t2 →  τ) by simp
  from wte1 wte2 eq EA
  show ∃τ″. S;Γ′ ⊢ ea : τ″ ∧ $ S τ″ = $ S τ by auto
qed

lemma stlc-implies-istlc-impl2:
  ⟦ idempotent S; $S Γ ⊢ $S e : $S τ ⟧ ⟹ (∃ τ′. S;Γ ⊢ e : τ′ ∧ $S τ′ = $S τ)
  using stlc-implies-istlc-impl by blast
```

```
lemma stlc-implies-istlc:
  assumes wte: $S Γ ⊢ $S e : τ and ids: idempotent S
  shows ∃ τ'. S;Γ ⊢ e : τ' ∧ $S τ' = τ
proof −
  from ids wte have wte2: $S Γ ⊢ $S e : $S τ by (rule ewt-ewSt)
  from ids wte2 obtain τ' where wtep: S;Γ ⊢ e : τ' and stst: $S τ' = $S τ
    using stlc-implies-istlc-impl2 by blast
  from ids wte have $S τ = τ by (rule ewt-steqt)
  with wtep stst show ?thesis by auto
qed

lemma stlc-wt-implies-teqSt:
   assumes idems: idempotent S and ewt: $S Γ ⊢ $S e : τ
   shows τ = $S τ
proof −
  from ewt idems have ∀ a. a ∈ FTV τ ⟶ %S a = UVarT a by (rule ftv-wt-sub)
  hence $S τ = τ using ftv-dom-id by blast
  thus τ = $S τ by auto
qed

lemma subst-const[simp]: $S (TypeOf c) = TypeOf c
  apply (case-tac c) apply auto done

lemma subst-env[rule-format]:
  ∀ x τ S. lookup Γ x = Some τ ⟶ lookup ($ S Γ) x = Some ($ S τ)
  apply (induct Γ) apply simp
  apply clarify apply (simp add: app-subst-pair)
    apply (case-tac a = x) apply simp
    apply auto
    done

lemma istlc-implies-stlc:
  S;Γ ⊢ e : τ ⟹ $S Γ ⊢ $S e : $S τ
  apply (induct rule: istlc-wt.induct)
  apply simp apply (rule Var) apply (simp add: subst-env)
  apply simp apply blast
  defer
  apply simp apply (rule App) apply simp apply simp
proof −
  fix S::subst and Γ::env and τ1 τ2 e x
  assume S;(x,τ1)#Γ ⊢ e : τ2 and SE: $ S ((x,τ1)#Γ) ⊢ $ S e : $ S τ2
  have $ S ((x,τ1)#Γ) = (x, $S τ1)#($ S Γ) by (simp add: app-subst-pair)
  with SE have (x,$S τ1)#($ S Γ) ⊢ $ S e : $ S τ2 by simp
  thus $S Γ ⊢ $ S (λx:τ1. e) : $ S (τ1 → τ2) by auto
qed

— Theorem 1 of the paper
theorem sltc-istlc-equivalent:
  (idempotent S ∧ $S Γ ⊢ $S e : τ ⟶ (∃ τ'. S;Γ ⊢ e : τ' ∧ $S τ' = τ))
   ∧ (S;Γ ⊢ e : τ ⟶ $S Γ ⊢ $S e : $S τ)
  apply (rule conjI)
  using stlc-implies-istlc apply simp
  using istlc-implies-stlc apply simp
  done
```

A.3 Choosing Fresh Variables

In various places within the formal development we need to choose a "fresh" variable. More specifically, we need to choose a variable that is not in some set, such as the domain of the type environment. Variables are represented here as natural numbers, and we constructively choose a fresh variable by taking the successor of the maximum number in the set. Of course, we must assume that the set in question is finite.

```
constdefs max :: nat ⇒ nat ⇒ nat
  max x y ≡ (if x < y then y else x)
declare max-def [simp]
```

To define the maximum number in a set, we take advantage of Isabelle's ability to fold over a finite set. To use fold with the above max function, we must first prove a few properties of max, but the proofs go through automatically.

```
interpretation AC-max: ACe [max 0::nat]
  by unfold-locales (auto intro: add-assoc add-commute)

constdefs setmax :: nat set ⇒ nat
  setmax S ≡ fold max (λ x. x) 0 S
```

We want to show that the successor of the maximum element of a set is not in the set. Towards proving that we prove the following lemma.

```
lemma max-ge: finite L ⟹ ∀ x ∈ L. x ≤ setmax L
  apply (induct rule: finite-induct)
  apply simp
  apply clarify
  apply (case-tac xa = x)
proof –
  fix x and F::nat set and xa
  assume fF: finite F and xF: x ∉ F and xax: xa = x
  from fF xF have mc: setmax (insert x F) = max x (setmax F)
    apply (simp only: setmax-def)
    apply (rule AC-max.fold-insert)
    apply auto done
  with xax show xa ≤ setmax (insert x F)
    apply clarify by simp
next
  fix x and F::nat set and xa
  assume fF: finite F and xF: x ∉ F
    and axF: ∀x∈F. x ≤ setmax F
    and xsxF: xa ∈ insert x F
    and xax: xa ≠ x
  from xax xsxF have xaF: xa ∈ F by auto
  with axF have xasF: xa ≤ setmax F by blast
  from fF xF have mc: setmax (insert x F) = max x (setmax F)
    apply (simp only: setmax-def)
    apply (rule AC-max.fold-insert)
    apply auto done
  with xasF show xa ≤ setmax (insert x F) by auto
qed

lemma max-is-fresh[simp]:
```

```
  assumes F: finite L shows Suc (setmax L) ∉ L
proof
  assume ssl: Suc (setmax L) ∈ L
  with F max-ge have Suc (setmax L) ≤ setmax L by blast
  thus False by simp
qed

lemma greaterthan-max-is-fresh[simp]:
  assumes F: finite L and I: setmax L < i
  shows i ∉ L
proof
  assume ssl: i ∈ L
  with F max-ge have i ≤ setmax L by blast
  with I show False by simp
qed

lemma subset-implies-lessmax-impl:
  finite A ⟹ ∀ B. finite B ∧ A ⊆ B ⟶ setmax A ≤ setmax B
  apply (induct rule: finite-induct)
  apply (simp add: setmax-def)
proof −
  fix x F assume fF: finite F and xF: x ∉ F
    and IH: ∀B. finite B ∧ F⊆B ⟶ setmax F ≤ setmax B
  show ∀B. finite B ∧ insert x F ⊆ B ⟶ setmax (insert x F) ≤ setmax B
  proof clarify
    fix B assume fB: finite B and xFsubB: insert x F ⊆ B
    from fF xF have smxF: setmax (insert x F) = max x (setmax F)
      apply (simp only: setmax-def) apply (rule AC-max.fold-insert) by auto
    from xFsubB have xB: x ∈ B by auto
    from fB xB have xleB: x ≤ setmax B using max-ge by blast
    from xFsubB have FsubB: F ⊆ B by auto
    from fB FsubB IH have setmax F ≤ setmax B by simp
    with xleB smxF show setmax (insert x F) ≤ setmax B by simp
  qed
qed

lemma subset-implies-lessmax:
  ⟦ finite B; A ⊆ B ⟧ ⟹ setmax A ≤ setmax B
  apply (frule finite-subset) apply simp
  using subset-implies-lessmax-impl apply simp
  done
```

A.4 The Consistency and Less Informative Relations

```
inductive consistent :: ty ⇒ ty ⇒ bool (infix ∼ 51)
where
  CRefl[intro!]: τ ∼ τ |
  CFun[intro!]: ⟦ σ ∼ τ; σ′ ∼ τ′ ⟧ ⟹ (σ → σ′) ∼ (τ → τ′) |
  CUnR[intro!]: τ ∼ ? |
  CUnL[intro!]: ? ∼ τ

lemma consistent-reflexive: σ ∼ σ
  apply (induct rule: ty.induct) apply auto done
```

```
lemma consistent-symmetric:  σ ∼ τ ⟹ τ ∼ σ
  apply (induct rule: consistent.induct) by auto

lemma consistent-not-trans:
  ¬ (∀ τ₁ τ₂ τ₃. τ₁ ∼ τ₂ ∧ τ₂ ∼ τ₃ ⟶ τ₁ ∼ τ₃)
proof −
  have A: IntT ∼ ? by auto
  have B: ? ∼ BoolT by auto
  have C: ¬ (IntT ∼ BoolT) by auto
  from A B C show ?thesis by auto
qed

inductive less-info :: ty ⇒ ty ⇒ bool (infixl ⊑ 51)
where
  LEInt[intro!]: IntT ⊑ IntT |
  LEBool[intro!]: BoolT ⊑ BoolT |
  LEUVar[intro!]: UVarT α ⊑ UVarT α |
  LEFun[intro!]: ⟦ σ ⊑ τ; σ′ ⊑ τ′ ⟧ ⟹ (σ → σ′) ⊑  (τ → τ′) |
  LEBottom[intro!]: ? ⊑ τ

lemma less-info-refl[intro!]: t ⊑ t
  apply (induct t) by auto

lemma less-info-transitive-impl:
  ∀ ϱ τ. ϱ ⊑ σ ∧ σ ⊑ τ ⟶ ϱ ⊑ τ
  apply (induct σ) apply blast+ done

lemma less-info-transitive: ⟦ ϱ ⊑ σ; σ ⊑ τ ⟧ ⟹ ϱ ⊑ τ
  using less-info-transitive-impl by blast

lemma less-info-implies-consistent: σ ⊑ τ ⟹ σ ∼ τ
  apply (induct rule: less-info.induct) by auto

lemma less-cons-implies-cons[rule-format]: σ ⊑ τ ⟹ (∀ τ′. τ ∼ τ′ ⟶ σ ∼ τ′)
  apply (induct rule: less-info.induct)
  apply simp
  apply simp
  apply simp
  apply clarify
    apply (erule cons-fun-any)
    apply simp
    apply (erule-tac x=τ in allE)
    apply (erule-tac x=τ′ in allE)
    apply force
    apply simp apply blast
  apply simp
    apply (erule-tac x=τ′′ in allE)
    apply (erule-tac x=τ′b in allE)
    apply force
  apply force
  done

lemma cons-less-less: t1 ∼ t2 ⟹ (∃ t3. t1 ⊑ t3 ∧ t2 ⊑ t3)
  apply (induct rule: consistent.induct) apply blast+ done
```

```
lemma less-less-cons[rule-format]: t1 ⊑ t2 ⟹ (∀ t3. t3 ⊑ t2 ⟶ t1 ∼ t3)
  apply (induct rule: less-info.induct)
  apply clarify apply (rule consistent-symmetric)
    apply (rule less-info-implies-consistent) apply assumption
  apply blast
  apply blast
  apply blast
  apply blast
  done
```

A.5 The Gradual Type System

```
inductive gtlc-wt :: env ⇒ [expr,ty] ⇒ bool (- ⊢g - : - [52,52,52] 51)
where
  WTVar[intro!]: ⟦ lookup Γ x = Some τ ⟧ ⟹ Γ ⊢g Var x : τ |

  WTConst[intro!]: Γ ⊢g Const c : TypeOf c |

  WTAbs[intro!]: ⟦ (x,τ)#Γ ⊢g e : ϱ ⟧ ⟹ Γ ⊢g (λx:τ. e) : τ → ϱ |

  WTApp1[intro!]: ⟦ Γ ⊢g e : ?; Γ ⊢g e′ : τ ⟧ ⟹ Γ ⊢g (App e e′) : ? |

  WTApp2[intro!]: ⟦ Γ ⊢g e : τ → ϱ; Γ ⊢g e′ : τ′; τ′ ∼ τ ⟧
           ⟹ Γ ⊢g (App e e′) : ϱ
```

A.6 Consistent-equal and Consistent-less

```
inductive consistent-equal :: subst ⇒ [ty, ty] ⇒ bool (- ⊢ - ≃ - [50,50,50] 51)
  and consistent-less :: subst ⇒ [ty, ty] ⇒ bool (- ⊢ - ⊑ - [50,50,50] 51)
where
  CEInt[intro!]: S ⊢ IntT ≃ IntT |
  CEBool[intro!]: S ⊢ BoolT ≃ BoolT |
  CEDynL[intro!]: S ⊢ ? ≃ τ |
  CEDynR[intro!]: S ⊢ τ ≃ ? |
  CEFun[intro!]: ⟦ S ⊢ σ ≃ τ; S ⊢ σ′ ≃ τ′ ⟧ ⟹ S ⊢ (σ → σ′) ≃ (τ → τ′) |
  CEVarR[intro!]: S ⊢ τ ⊑ %S α ⟹ S ⊢ τ ≃ UVarT α |
  CEVarL[intro!]: S ⊢ τ ⊑ %S α ⟹ S ⊢ UVarT α ≃ τ |

  CLVar[intro!]: %S α = τ ⟹ S ⊢ UVarT α ⊑ τ |
  CLInt[intro!]: S ⊢ IntT ⊑ IntT |
  CLBool[intro!]: S ⊢ BoolT ⊑ BoolT |
  CLDynL[intro!]: S ⊢ ? ⊑ τ |
  CLFun[intro!]: ⟦ S ⊢ σ ⊑ τ; S ⊢ σ′ ⊑ τ′ ⟧ ⟹ S ⊢ (σ → σ′) ⊑ (τ → τ′)
```

A.6.1 Properties of Consistent-equal/less

The following lemmas correspond to Proposition 1 of the paper.

```
lemma consless-refl: S ⊢ t ⊑ $S t apply (induct t) by auto

lemma consless-trans-impl: (S ⊢ τ1 ≃ τ2 ⟶ True)
  ∧ (S ⊢ τ2 ⊑ τ3 ⟶ (∀ τ1. τ1 ⊑ τ2 ⟶ S ⊢ τ1 ⊑ τ3))
  apply (induct rule: consistent-equal-consistent-less.induct) by auto
lemma consless-trans: ⟦ τ1 ⊑ τ2; S ⊢ τ2 ⊑ τ3 ⟧ ⟹ S ⊢ τ1 ⊑ τ3
  using consless-trans-impl by blast
```

```
lemma conseq-refl: S ⊢ τ ≃ τ
  apply (induct τ) apply blast+ done

lemma conseq-symm-impl: (S ⊢ τ ≃ τ′ ⟶ S ⊢ τ′ ≃ τ) ∧ (S ⊢ τ ⊑ τ′ ⟶ True)
  apply (induct rule: consistent-equal-consistent-less.induct)
  apply auto
  done

lemma conseq-symm: S ⊢ τ ≃ τ′ ⟹ S ⊢ τ′ ≃ τ
  using conseq-symm-impl by blast

lemma conseq-less-no-dyn-equal-impl:
  (S ⊢ τ1 ≃ τ2 ⟶ no-dyn τ1 ∧ no-dyn τ2 ⟶ $S τ1 = $S τ2)
  ∧ (S ⊢ τ ⊑ τ′ ⟶ no-dyn τ ⟶ $S τ = τ′)
  apply (induct rule: consistent-equal-consistent-less.induct)
  apply simp+
  done

lemma conseq-no-dyn-equal:
  ⟦ S ⊢ τ ≃ τ′; no-dyn τ; no-dyn τ′ ⟧ ⟹ $S τ = $S τ′
  using conseq-less-no-dyn-equal-impl by blast

lemma less-no-dyn-equal:
  ⟦ S ⊢ τ ⊑ τ′; no-dyn τ ⟧ ⟹ $S τ = τ′
  using conseq-less-no-dyn-equal-impl by blast

lemma less-conseq-less-impl: (S ⊢ τ1 ≃ τ2 ⟶ True)
  ∧ (S ⊢ τ ⊑ τ′′ ⟶ no-dyn τ ⟶
  (∀ τ′. no-dyn τ′ ∧ S ⊢ τ ≃ τ′ ⟶ S ⊢ τ′ ⊑ τ′′))
  apply (induct rule: consistent-equal-consistent-less.induct)
  apply simp apply simp apply simp apply simp apply simp apply simp apply simp
  apply force apply force apply force apply force apply (rule impI) apply simp
  apply (case-tac no-dyn σ) apply simp
  prefer 2 apply simp
  apply clarify
  apply (erule conseq-fun-any)
    apply simp apply simp
    apply (case-tac no-dyn τ′′) apply simp
    prefer 2 apply simp
    apply (rule CLFun)
    apply force
    apply force
    apply simp
      apply (rule CLVar)
      apply (erule consless-fun-any)
      apply simp
      apply (rule conjI)
proof −
  fix S σ σ′ τ τ′ τ′a α τ′′ τ′b
  assume st: S ⊢ σ ⊑ τ and ns: no-dyn σ and stt: S ⊢ σ ⊑ τ′′
  from st ns have sst: $S σ = τ by (rule less-no-dyn-equal)
  from stt ns have $S σ = τ′′ by (rule less-no-dyn-equal)
```

with *sst* **show** $\tau'' = \tau$ **by** *simp*
next
fix $S\ \sigma\ \sigma'\ \tau\ \tau'\ \tau'a\ \alpha\ \tau''\ \tau'b$
assume *st*: $S \vdash \sigma' \sqsubseteq \tau'$ **and** *ns*: *no-dyn* σ' **and** *stt*: $S \vdash \sigma' \sqsubseteq \tau'b$
from *st ns* **have** *sst*: $\$S\ \sigma' = \tau'$ **by** (*rule less-no-dyn-equal*)
from *stt ns* **have** $\$S\ \sigma' = \tau'b$ **by** (*rule less-no-dyn-equal*)
with *sst* **show** $\tau'b = \tau'$ **by** *simp*
qed

lemma *less-conseq-less*: $[\![\ S \vdash \tau \sqsubseteq \tau''$; *no-dyn* τ; *no-dyn* τ'; $S \vdash \tau \simeq \tau'\ ]\!]$
$\Longrightarrow S \vdash \tau' \sqsubseteq \tau''$
using *less-conseq-less-impl* **by** *blast*

lemma *less-less-conseq-impl*:
$(S \vdash \tau \simeq \tau' \longrightarrow \mathit{True}) \wedge$
$(S \vdash \tau \sqsubseteq \varrho \longrightarrow (\forall\ \tau'.\ S \vdash \tau' \sqsubseteq \varrho \longrightarrow S \vdash \tau \simeq \tau'))$
apply (*induct rule*: *consistent-equal-consistent-less.induct*)
apply *simp+*
apply *blast*
apply *force*
apply *force*
apply *force*
apply *clarify* **apply** (*rule consless-any-fun*) **apply** *auto*
done

lemma *less-less-conseq*:
$[\![\ S \vdash \tau \sqsubseteq \varrho;\ S \vdash \tau' \sqsubseteq \varrho\ ]\!] \Longrightarrow S \vdash \tau \simeq \tau'$
using *less-less-conseq-impl* **by** *blast*

lemma *subst-typeof*: $\$\ S\ (\mathit{TypeOf}\ c) = \mathit{TypeOf}\ c$
apply (*case-tac c*) **apply** *auto* **done**

lemma *subst-const*: $\$\ S\ (\mathit{Const}\ c) = \mathit{Const}\ c$
apply (*case-tac c*) **apply** *auto* **done**

lemma *subst-extend-env*: $\$\ S\ ((x,\tau)\#\Gamma) = (x,\ \$S\ \tau)\#(\$\ S\ \Gamma)$
by (*simp add*: *app-subst-pair*)

lemma *cons-any-fun2*:
$\tau \sim t1 \rightarrow t2 \Longrightarrow (\tau = ?) \vee (\exists\ s1\ s2.\ \tau = s1 \rightarrow s2 \wedge s1 \sim t1 \wedge s2 \sim t2)$
using *cons-any-fun* **by** *blast*

lemma *ce-less-implies-cons-less*:
$(S \vdash \tau \simeq \tau' \longrightarrow \$S\ \tau \sim \$S\ \tau') \wedge (S \vdash \tau \sqsubseteq \tau' \longrightarrow \$S\ \tau \sqsubseteq \tau')$
apply (*induct rule*: *consistent-equal-consistent-less.induct*)
apply *force* **apply** *force* **apply** *force* **apply** *force*
apply *simp* **apply** (*rule CFun*) **apply** *simp* **apply** *simp*
apply *simp* **using** *less-info-implies-consistent* **apply** *blast*
apply *simp*
apply (*frule less-info-implies-consistent*)
apply (*frule consistent-symmetric*) **apply** *simp*
apply *force+*
done

lemma *cons-eq-implies-cons*:

```
  S ⊢ τ ≃ τ' ⟹ $S τ ∼ $S τ'
  using ce-less-implies-cons-less by blast

lemma cons-less-implies-less:
  S ⊢ τ ⊑ τ' ⟹ $S τ ⊑ τ'
  using ce-less-implies-cons-less by blast

lemma conseq-any-fun-var:
  S ⊢ τ ≃ τ' → UVarT β ⟹
    τ = ? ∨ (∃ t1 t2. $S τ = t1 → t2 ∧ t1 ∼ $S τ' ∧ t2 ⊑ %S β)
  apply (case-tac τ)
  defer
  apply force
  apply force
  apply simp
  apply simp
  apply (erule conseq-fun-fun) apply (rule conjI)
  apply (rule cons-eq-implies-cons) apply simp
  apply (erule conseq-any-uvar)
    apply simp apply force
    apply (rule cons-less-implies-less) apply simp
    apply simp apply (erule consless-uvar-any) apply force
proof −
  fix α assume ttb: S ⊢ τ ≃ τ' → UVarT β and t: τ = UVarT α
  from ttb t have tba: S ⊢ τ' → UVarT β ⊑ %S α
    apply simp apply (erule conseq-uvar-fun) by blast
  from tba obtain t1 t2 where tt1: S ⊢ τ' ⊑ t1  and bt2: S ⊢ UVarT β ⊑ t2
    and sa: %S α = t1 → t2 using consless-fun-var-any by blast
  from tt1 have $ S τ' ⊑ t1 by (rule cons-less-implies-less)
  hence t1t: t1 ∼ $ S τ' using less-info-implies-consistent consistent-symmetric by blast
  from bt2 have %S β = t2 by force
  hence t2sb: t2 ⊑ %S β by force
  from t sa t1t t2sb
  show τ = ? ∨ (∃ t1 t2. $S τ = t1 → t2 ∧ t1 ∼ $S τ' ∧ t2 ⊑ %S β) by simp
qed

lemma conseq-any-fun-var-rule:
  ⟦ S ⊢ τ ≃ τ' → UVarT β;
    τ = ? ⟹ P;
    ⋀ t1 t2. ⟦ $S τ = t1 → t2; t1 ∼ $S τ'; t2 ⊑ %S β ⟧ ⟹ P⟧
   ⟹ P
  apply (frule conseq-any-fun-var) apply (erule disjE)
  apply simp
  apply (erule exE)+ apply simp apply clarify
  apply (case-tac $S τ) apply force apply force apply force apply force
  apply simp apply clarify apply simp
proof −
  fix t1 t2
  assume A: ⋀t1a t2a. ⟦t1 = t1a ∧ t2 = t2a; t1a ∼ $ S τ'; t2a ⊑ %S β⟧ ⟹ P
    and B: t1 ∼ $ S τ' and C: t2 ⊑ %S β
    from A[of t1 t2] B C show P apply blast done
qed

lemma prop1-item-6-and-7-fwd:
  (S ⊢ t1 ≃ t2 ⟶ FTV t1 = {} ∧ FTV t2 = {} ⟶ t1 ∼ t2)
```

$\wedge\ (S \vdash \mathit{t1} \sqsubseteq \mathit{t2} \longrightarrow \mathit{FTV}\ \mathit{t1} = \{\} \longrightarrow \mathit{t1} \sqsubseteq \mathit{t2})$
apply (*induct rule*: *consistent-equal-consistent-less.induct*)
apply *blast* **apply** *blast* **apply** *blast* **apply** *blast*
apply *clarify* **apply** *simp* **apply** (*rule CFun*) **apply** *simp* **apply** *simp*
apply *simp* **apply** *simp* **apply** *simp* **apply** *blast* **apply** *blast* **apply** *blast*
apply *clarify* **apply** (*rule LEFun*) **apply** *simp* **apply** *simp*
done

lemma *prop1-item-6-back*:
$\mathit{t1} \sim \mathit{t2} \Longrightarrow \mathit{FTV}\ \mathit{t1} = \{\} \wedge \mathit{FTV}\ \mathit{t2} = \{\} \longrightarrow (\forall\ S.\ S \vdash \mathit{t1} \simeq \mathit{t2})$
apply (*induct rule*: *consistent.induct*)
apply *clarify* **apply** (*rule conseq-refl*)
apply *clarify* **apply** (*rule CEFun*) **apply** *simp* **apply** *simp*
apply *blast* **apply** *blast* **done**

lemma *ftv-empty-subst-id*[*rule-format*]:
$\forall\ S.\ \mathit{FTV}\ \tau = \{\} \longrightarrow \$S\ \tau = (\tau{::}\mathit{ty})$
apply (*induct* τ) **by** *auto*

lemma *prop1-item-7-back*:
$\mathit{t1} \sqsubseteq \mathit{t2} \Longrightarrow \mathit{FTV}\ \mathit{t1} = \{\} \longrightarrow (\forall\ S.\ S \vdash \mathit{t1} \sqsubseteq \mathit{t2})$
apply (*induct rule*: *less-info.induct*)
apply *blast*
apply *blast*
defer
apply *clarify* **apply** (*rule CLFun*) **apply** *simp* **apply** *simp*
apply *blast*
apply *auto*
done

lemma *widen-conseq-consless*:
$(S \vdash \tau_1 \simeq \tau_2 \longrightarrow (\forall\alpha.\ \alpha \in \mathit{FTV}\ \tau_1 \cup \mathit{FTV}\ \tau_2 \longrightarrow \%S'\ \alpha = \%S\ \alpha)$
$\quad\longrightarrow S' \vdash \tau_1 \simeq \tau_2)$
$\wedge\ (S \vdash \tau_1 \sqsubseteq \tau_2 \longrightarrow (\forall\alpha.\ \alpha \in \mathit{FTV}\ \tau_1 \longrightarrow \%S'\ \alpha = \%S\ \alpha)$
$\quad\longrightarrow S' \vdash \tau_1 \sqsubseteq \tau_2)$
apply (*induct rule*: *consistent-equal-consistent-less.induct*)
apply *force*+ **done**

lemma *widen-conseq*:
$[\![\ S \vdash \tau_1 \simeq \tau_2;\ (\forall\alpha.\ \alpha \in \mathit{FTV}\ \tau_1 \cup \mathit{FTV}\ \tau_2 \longrightarrow \%S'\ \alpha = \%S\ \alpha)\ ]\!]$
$\Longrightarrow S' \vdash \tau_1 \simeq \tau_2$
using *widen-conseq-consless* **by** *blast*

lemma *widen-consless*:
$[\![\ S \vdash \tau_1 \sqsubseteq \tau_2;\ (\forall\alpha.\ \alpha \in \mathit{FTV}\ \tau_1 \longrightarrow \%S'\ \alpha = \%S\ \alpha)\ ]\!] \Longrightarrow S' \vdash \tau_1 \sqsubseteq \tau_2$
using *widen-conseq-consless* **by** *blast*

A.7 The Gradual Type System with Type Variables

inductive *igtlc-wt* :: [*subst*,*env*,*nat*,*nat*] $\Rightarrow$ [*expr*,*ty*] $\Rightarrow$ *bool* (-;-;-;- $\vdash_g$ - : - [*52*,*52*,*52*,*52*,*52*,*52*] *51*)
where
GVar[*intro*!]: $[\![\ \mathit{lookup}\ \Gamma\ x = \mathit{Some}\ \tau\ ]\!] \Longrightarrow S;\Gamma;n;n \vdash_g \mathit{Var}\ x : \tau\ |$

GConst[*intro*!]: $S;\Gamma;n;n \vdash_g \mathit{Const}\ c : \mathit{TypeOf}\ c\ |$

GAbs[*intro*!]: $[\![\ S;(x,\tau_1)\#\Gamma;m;n \vdash_g e : \tau_2\ ]\!] \Longrightarrow S;\Gamma;m;n \vdash_g (\lambda x{:}\tau_1.\ e) : \tau_1 \rightarrow \tau_2\ |$

```
  GApp[intro!]: ⟦ S;Γ;n0;n1 ⊢g e : τ1; S;Γ;n1;n2 ⊢g e' : τ2;
                  S ⊢ τ1 ≃ (τ2 → UVarT n2) ⟧
           ⟹ S;Γ;n0;Suc n2 ⊢g (App e e') : UVarT n2
lemma ftv-env-ftv-ty[rule-format]:
  ∀ x τ. lookup Γ x = Some τ ⟶ FTV τ ⊆ FTV Γ
  apply (induct Γ) by auto

lemma igtlc-fresh-grows:
  S;Γ;m;n ⊢g e : τ ⟹ m ≤ n
  apply (induct rule: igtlc-wt.induct)
  apply simp+ done

lemma igtlc-ftv-result:
  S;Γ;m;n ⊢g e : τ ⟹ (∀ α. α ∈ FTV τ ⟶ α ∈ FTV Γ ∪ FTV e ∨ (m ≤ α ∧ α < n))
  (is S;Γ;m;n ⊢g e : τ ⟹ ?P S Γ m n e τ)
  apply (induct rule: igtlc-wt.induct)
  apply clarify apply simp using ftv-env-ftv-ty apply blast
  apply simp apply clarify apply (case-tac c) apply simp apply simp apply simp apply simp
proof −
  fix S Γ τ1 τ2 e m n x
  assume S;(x, τ1)#Γ;m;n ⊢g e : τ2
    and IH: ∀α. α ∈ FTV τ2 ⟶ α ∈ FTV ((x, τ1)#Γ) ∪ FTV e ∨ m ≤ α ∧ α < n
  show ?P S Γ m n (λx:τ1. e) (τ1 → τ2)
    apply (rule allI) apply (rule impI)
  proof −
    fix α assume af12: α ∈ FTV (τ1 → τ2)
    from af12 have α ∈ FTV τ1 ∨ α ∈ FTV τ2 by simp
    moreover { assume af1: α ∈ FTV τ1
      from af1 have α ∈ FTV (λx:τ1. e) by simp
      hence α ∈ FTV Γ ∪ FTV (λx:τ1. e) ∨ m ≤ α ∧ α < n by simp
    } moreover { assume af2: α ∈ FTV τ2
      from af2 IH have α ∈ FTV ((x,τ1)#Γ) ∪ FTV e ∨ m ≤ α ∧ α < n by simp
      moreover { assume α ∈ FTV ((x, τ1)#Γ)
        hence α ∈ FTV Γ ∨ α ∈ FTV τ1 apply (simp add: FTV-pair) by blast
        hence α ∈ FTV Γ ∪ FTV (λx:τ1. e) ∨ m ≤ α ∧ α < n by force
      } moreover { assume α ∈ FTV e
        hence α ∈ FTV (λx:τ1. e) by simp
        hence α ∈ FTV Γ ∪ FTV (λx:τ1. e) ∨ m ≤ α ∧ α < n by simp
      } moreover { assume m ≤ α ∧ α < n
        hence α ∈ FTV Γ ∪ FTV (λx:τ1. e) ∨ m ≤ α ∧ α < n by simp
      } ultimately have α ∈ FTV Γ ∪ FTV (λx:τ1. e) ∨ m ≤ α ∧ α < n by blast
    } ultimately show α ∈ FTV Γ ∪ FTV (λx:τ1. e) ∨ m ≤ α ∧ α < n by blast
  qed
next
  fix S Γ τ1 τ2 e e' n0 n1 n2
  assume wte: S;Γ;n0;n1 ⊢g e : τ1
    and IH1: ∀α. α ∈ FTV τ1 ⟶ α ∈ FTV Γ ∪ FTV e ∨ n0 ≤ α ∧ α < n1
    and wtep: S;Γ;n1;n2 ⊢g e' : τ2
    and IH2: ∀α. α ∈ FTV τ2 ⟶ α ∈ FTV Γ ∪ FTV e' ∨ n1 ≤ α ∧ α < n2
    and s12b: S ⊢ τ1 ≃ τ2 → UVarT n2
  show ∀α. α ∈ FTV (UVarT n2) ⟶ α ∈ FTV Γ ∪ FTV (App e e') ∨ n0 ≤ α ∧ α < Suc n2
    apply (rule allI) apply (rule impI)
  proof −
    fix α assume α ∈ FTV (UVarT n2) hence an2: α = n2 by simp
```

```
    from an2 have asn2: α < Suc n2 by simp
    from wte have n0n1: n0 ≤ n1 by (rule igtlc-fresh-grows)
    from wtep have n1n2: n1 ≤ n2 by (rule igtlc-fresh-grows)
    from n0n1 n1n2 an2 have n0a: n0 ≤ α by simp
    from n0a asn2
    show α ∈ FTV Γ ∪ FTV (App e e') ∨ n0 ≤ α ∧ α < Suc n2 by simp
  qed
qed

lemma widen-subst-impl:
  S;Γ;m;n ⊢g e : τ ⟹
   (∀ α. α ∈ FTV Γ ∪ FTV e ∨ (m ≤ α ∧ α < n) ⟶ %S' α = %S α)
   ⟶ S';Γ;m;n ⊢g e : τ
  (is S;Γ;m;n ⊢g e : τ ⟹ ?P S Γ m n e τ)
  apply (induct rule: igtlc-wt.induct)
  apply force
  apply force
proof –
  fix S Γ τ1 τ2 e m n x
  assume S;(x,τ1)#Γ;m;n ⊢g e : τ2
    and IH: (∀α. α ∈ FTV ((x,τ1)#Γ) ∪ FTV e ∨ m ≤ α ∧ α < n ⟶ %S' α = %S α) ⟶
        S';(x, τ1)#Γ;m;n ⊢g e : τ2
  show (∀α. α ∈ FTV Γ ∪ FTV (λx:τ1. e) ∨ m ≤ α ∧ α < n ⟶ %S' α = %S α) ⟶
          S';Γ;m;n ⊢g λx:τ1. e : τ1 → τ2
  proof clarify
    assume fl: ∀α. α ∈ FTV Γ ∪ FTV (λx:τ1. e) ∨ m ≤ α ∧ α < n ⟶ %S' α = %S α
    from fl have (∀α. α ∈ FTV ((x, τ1)#Γ) ∪ FTV e ∨ m ≤ α ∧ α < n ⟶ %S' α = %S α)
      apply clarify apply (erule disjE)
        apply simp apply (erule disjE)
          apply (erule-tac x=α in allE) apply (simp add: FTV-pair)
          apply blast
        apply blast
      done
    with IH show S';(x, τ1)#Γ;m;n ⊢g e : τ2 by simp
  qed
next
  fix S Γ τ1 τ2 e e' n0 n1 n2
  assume wte: S;Γ;n0;n1 ⊢g e : τ1
    and IH1: ?P S Γ n0 n1 e τ1
    and wtep: S;Γ;n1;n2 ⊢g e' : τ2
    and IH2: ?P S Γ n1 n2 e' τ2
    and st12b: S ⊢ τ1 ≃ τ2 → UVarT n2
  show ?P S Γ n0 (Suc n2) (App e e') (UVarT n2)
  proof clarify
    assume fa: ∀α. α ∈ FTV Γ ∪ FTV (App e e') ∨ n0 ≤ α ∧ α < Suc n2 ⟶ %S' α = %S α
    from wte have n0n1: n0 ≤ n1 by (rule igtlc-fresh-grows)
    from wtep have n1n2: n1 ≤ n2 by (rule igtlc-fresh-grows)
    from fa n1n2
    have fe: (∀α. α ∈ FTV Γ ∪ FTV e ∨ n0 ≤ α ∧ α < n1 ⟶ %S' α = %S α) by simp
    from fe IH1 have wte2: S';Γ;n0;n1 ⊢g e : τ1 by simp
    from fa n0n1
    have fep: (∀α. α ∈ FTV Γ ∪ FTV e' ∨ n1 ≤ α ∧ α < n2 ⟶ %S' α = %S α) by simp
    from fep IH2 have wtep2: S';Γ;n1;n2 ⊢g e' : τ2 by simp
    from wte2 fe have aft1: (∀ α. α ∈ FTV τ1 ⟶ %S' α = %S α)
      using igtlc-ftv-result apply blast done
```

```
    from wtep2 fep have aft2: (∀ α. α ∈ FTV τ_2 ⟶ %S' α = %S α)
      using igtlc-ftv-result apply blast done
    from wte have n0n1: n0 ≤ n1 by (rule igtlc-fresh-grows)
    from wtep have n1n2: n1 ≤ n2 by (rule igtlc-fresh-grows)
    from fa n0n1 n1n2 have (∀ α. α ∈ FTV (UVarT n2) ⟶ %S' α = %S α) by auto
    with aft1 aft2
    have aft12b: (∀ α. α ∈ FTV τ_1 ∪ FTV (τ_2 → UVarT n2) ⟶ %S' α = %S α) by simp
    from st12b aft12b have S' ⊢ τ_1 ≃ τ_2 → UVarT n2 by (rule widen-conseq)
    with wte2 wtep2 show S';Γ;n0;Suc n2 ⊢_g App e e' : UVarT n2 by blast
  qed
qed

lemma widen-subst:
  ⟦ S;Γ;m;n ⊢_g e : τ; (∀ α. α ∈ FTV Γ ∪ FTV e ∨ (m ≤ α ∧ α < n) ⟶ %S' α = %S α) ⟧
    ⟹ S';Γ;m;n ⊢_g e : τ
  using widen-subst-impl by blast

— Theorem 2
theorem igtlc-implies-gtlc:
  S;Γ;m;n ⊢_g e : τ ⟹ FTV Γ = {} ∧ FTV e = {}
   ⟶ (∃ τ'. Γ ⊢_g e : τ' ∧ τ' ⊑ $S τ)
  (is S;Γ;m;n ⊢_g e : τ ⟹ ?P S Γ e τ)
proof (induct rule: igtlc-wt.induct)
  fix Γ::env and τ and x and S n assume gx: lookup Γ x = Some τ
  show ?P S Γ (Var x) τ
  proof clarify
    assume fg: FTV Γ = {}
    from gx fg have FTV τ = {} using ftv-env-ftv-ty by blast
    hence ∀ a. a ∈ FTV τ ⟶ %S a = UVarT a by simp
    hence $S τ = τ using ftv-dom-id by blast
    hence τ ⊑ $S τ apply simp by (rule less-info-refl)
    with gx show ∃τ'. Γ ⊢_g Var x : τ' ∧ τ' ⊑ $ S τ by blast
  qed
next
  fix S Γ c show ?P S Γ (Const c) (TypeOf c) by auto
next
  fix S::subst and x τ_1 and Γ::env and m n e τ_2
  assume IH: ?P S ((x,τ_1)#Γ) e τ_2
  show ?P S Γ (λx:τ_1. e) (τ_1 → τ_2)
  proof clarify
    assume fg: FTV Γ = {} and fl: FTV (λx:τ_1. e) = {}
    from fl have ft: FTV τ_1 = {} by simp
    with fg have fg2: FTV ((x,τ_1)#Γ) = {} by (simp add: FTV-fun)
    from fl have fe: FTV e = {} by simp
    from fg2 fe IH obtain τ' where wte: (x,τ_1)#Γ ⊢_g e : τ'
      and tpst2: τ' ⊑ $S τ_2 by blast
    from ft have ∀ a. a ∈ FTV τ_1 ⟶ %S a = UVarT a by simp
    hence $S τ_1 = τ_1 using ftv-dom-id by blast
    with tpst2 have stt: τ_1 → τ' ⊑ $S (τ_1 → τ_2)
      apply simp apply (rule LEFun) apply (rule less-info-refl) apply simp done
    from wte have Γ ⊢_g λx:τ_1. e : τ_1 → τ' by blast
    with stt show ∃τ'. Γ ⊢_g λx:τ_1. e : τ' ∧ τ' ⊑ $S (τ_1 → τ_2)
      apply (rule-tac x=τ_1 → τ' in exI)
      apply simp done
  qed
```

```
next
  fix S Γ n0 n1 e τ1 n2 e' τ2
  assume IH1: ?P S Γ e τ1
    and IH2: ?P S Γ e' τ2
    and st12b: S ⊢ τ1 ≃ τ2 → UVarT n2
  show ?P S Γ (App e e') (UVarT n2)
  proof clarify
    assume fg: FTV Γ = {} and fa: FTV (App e e') = {}
    from fa have fe: FTV e = {} by simp
    with fg IH1 obtain t1 where wte: Γ ⊢g e : t1 and t11: t1 ⊑ $S τ1 by blast
    from fa have fep: FTV e' = {} by simp
    with fg IH2 obtain t2 where wtep: Γ ⊢g e' : t2
      and st2: t2 ⊑ $S τ2 by blast
    from st12b show ∃τ'. Γ ⊢g App e e' : τ' ∧ τ' ⊑ $ S (UVarT n2)
    proof (rule conseq-any-fun-var-rule)
      assume t1d: τ1 = ?
      with t11 have T1d: t1 = ? by auto
      from wte T1d have wte2: Γ ⊢g e : ? by simp
      with wtep have A: Γ ⊢g App e e' : ? by blast
      have B: ? ⊑ $S (UVarT n2) by blast
      from A B show ?thesis by blast
    next
      fix t11 t12 assume st1: $ S τ1 = t11 → t12 and t11st2: t11 ∼ $ S τ2
        and t2sb: t12 ⊑ %S n2
      from t11 st1 have t1-le-t12: t1 ⊑ t11 → t12 by simp
      from t11st2 have st2t11: $ S τ2 ∼ t11 by (rule consistent-symmetric)
      from t1-le-t12 show ?thesis
      proof (rule le-any-fun)
        fix σ σ' assume st11: σ ⊑ t11 and spt12: σ' ⊑ t12
          and T1: t1 = σ → σ'
        with wte have wte2: Γ ⊢g e : σ → σ' by simp
        from st2 st2t11 have t2t11: t2 ∼ t11 by (rule less-cons-implies-cons)
        hence t11t2: t11 ∼ t2 by (rule consistent-symmetric)
        from st11 t11t2 have σ ∼ t2 by (rule less-cons-implies-cons)
        hence t2s: t2 ∼ σ by (rule consistent-symmetric)
        from wte2 wtep t2s have A: Γ ⊢g App e e' : σ' by blast
        from spt12 t2sb have σ' ⊑ %S n2 by (rule less-info-transitive)
        hence B: σ' ⊑ $S (UVarT n2) by simp
        from A B show ∃τ'. Γ ⊢g App e e' : τ' ∧ τ' ⊑ $ S (UVarT n2) by blast
      next
        assume T1d: t1 = ?
        from wte T1d have wte2: Γ ⊢g e : ? by simp
        with wtep have A: Γ ⊢g App e e' : ? by blast
        have B: ? ⊑ $S (UVarT n2) by blast
        from A B show ?thesis by blast
      qed
    qed
  qed
qed


lemma lookup-subst-subst:
  lookup Γ x = Some τ ⟹ lookup ($ S Γ) x = Some ($ S τ)
  apply (induct Γ)
  apply simp
```

```
  apply (case-tac a) apply (case-tac aa = x)
    apply (simp add: app-subst-pair)
    apply (simp add: app-subst-pair)
  done

— Part 1 of Theorem 3 of the paper
theorem igtlc-wt-implies-gtlc:
  S;Γ;m;n ⊢g e : τ ⟹ (∃ τ'. $S Γ ⊢g $S e : τ' ∧ τ' ⊑ $S τ)
proof (induct rule: igtlc-wt.induct)
  fix Γ::env and τ x S n assume lookup Γ x = Some τ
  thus ∃τ'. $ S Γ ⊢g $ S (Var x) : τ' ∧ τ' ⊑ $ S τ
    apply simp apply (rule-tac x=$S τ in exI) apply (rule conjI)
    apply (rule WTVar) apply (simp add: lookup-subst-subst) apply blast done
next
  fix S Γ n c show ∃τ'. $ S Γ ⊢g $ S (Const c) : τ' ∧ τ' ⊑ $ S (TypeOf c)
    apply simp apply blast done
next
  fix S x τ1 Γ m n e τ2
  assume IH: ∃τ'. $ S ((x,τ1)#Γ) ⊢g $ S e : τ' ∧ τ' ⊑ $ S τ2
  from IH obtain τ' where wte: $ S ((x,τ1)#Γ) ⊢g $ S e : τ'
    and tst: τ' ⊑ $ S τ2 by blast
  from wte have (x,$S τ1)#($ S Γ) ⊢g $ S e : τ' by (simp only: subst-extend-env)
  hence wtl: $S Γ ⊢g (λx:$S τ1. $S e) : ($S τ1) → τ' by blast
  from tst have $S τ1 → τ' ⊑ $S (τ1 → τ2) by auto
  with wtl show ∃τ'. $ S Γ ⊢g $ S (λx:τ1. e) : τ' ∧ τ' ⊑ $ S (τ1 → τ2)
    apply (rule-tac x=$S τ1 → τ' in exI) by auto
next
  fix S Γ n0 n1 e τ1 n2 e' τ2
  assume IH1: ∃τ'. $ S Γ ⊢g $ S e : τ' ∧ τ' ⊑ $ S τ1
    and IH2: ∃τ'. $ S Γ ⊢g $ S e' : τ' ∧ τ' ⊑ $ S τ2
    and t123: S ⊢ τ1 ≃ τ2 → UVarT n2
  from IH1 obtain t1' where wte: $ S Γ ⊢g $ S e : t1'
    and t1st: t1' ⊑ $ S τ1 by blast
  from IH2 obtain t2' where wtep: $ S Γ ⊢g $ S e' : t2'
    and tpst2: t2' ⊑ $ S τ2 by blast
  from t123 show ∃τ'. $ S Γ ⊢g $ S (App e e') : τ' ∧ τ' ⊑ $ S (UVarT n2)
  proof (rule conseq-any-fun-var-rule)
    assume t1: τ1 = ?
    from t1 t1st have t1p: t1' = ? by auto
    with wte have wte: $ S Γ ⊢g $ S e : ? by simp
    from wte wtep have $S Γ ⊢g App ($S e) ($S e') : ? by (rule WTApp1)
    thus ∃τ'. $ S Γ ⊢g $ S (App e e') : τ' ∧ τ' ⊑ $ S (UVarT n2) apply simp by blast
  next
    fix t1 t2
    assume st1: $ S τ1 = t1 → t2 and t1s: t1 ∼ $ S τ2 and t2b: t2 ⊑ %S n2
    from t1st st1 have t1p12: t1' ⊑ t1 → t2 by simp
    hence t1' = ? ∨ (∃ t11 t12. t1' = t11 → t12) using le-any-fun by blast
    moreover { assume t1p: t1' = ?
      with wte have wte: $ S Γ ⊢g $ S e : ? by simp
      from wte wtep have $S Γ ⊢g App ($S e) ($S e') : ? by (rule WTApp1)
      hence ∃τ'. $ S Γ ⊢g $ S (App e e') : τ' ∧ τ' ⊑ $ S (UVarT n2)
        apply simp by blast
    } moreover { assume X: ∃ t11 t12. t1' = t11 → t12
      from X obtain t11 t12 where T1: t1' = t11 → t12 by blast
      from T1 t1p12 have t11t1: t11 ⊑ t1 by blast
```

```
      from T1 t1p12 have t12t2: t12 ⊑ t2 by blast
      from wte T1 have wte2: $ S Γ ⊢g $ S e : t11 → t12 by simp
      from t11t1 t1s have t11st2: t11 ∼ $S τ2 by (rule less-cons-implies-cons)
      hence st2t11: $S τ2 ∼ t11 by (rule consistent-symmetric)
      from tpst2 st2t11 have t2t11: t2′ ∼ t11 by (rule less-cons-implies-cons)
      from wte2 wtep t2t11 have wta: $S Γ ⊢g App ($S e) ($S e′) : t12
        by (rule WTApp2)
      from t12t2 t2b have t12b: t12 ⊑ %S n2 by (rule less-info-transitive)
      from wta t12b
      have ∃τ′. $ S Γ ⊢g $ S (App e e′) : τ′ ∧ τ′ ⊑ $ S (UVarT n2)
        apply simp by blast
    } ultimately show ∃τ′. $ S Γ ⊢g $ S (App e e′) : τ′ ∧ τ′ ⊑ $ S (UVarT n2)
      by blast
  qed
qed

lemma no-dyn-lookup:
  ⋀ x τ. ⟦ lookup Γ x = Some τ; no-dyn Γ ⟧ ⟹ no-dyn τ
  apply (induct Γ)
  apply simp
  apply simp apply (case-tac a) apply simp
    apply (case-tac no-dyn b) apply simp
    apply (case-tac aa = x) apply simp
    apply simp apply simp
  done

— This is Theorem 4 of the paper
theorem igtlc-wt-no-dyn-implies-istlc:
  S;Γ;m;n ⊢g e : τ ⟹ no-dyn Γ ∧ no-dyn e ⟶ S;Γ ⊢ e : τ ∧ no-dyn τ
  apply (induct rule: igtlc-wt.induct)
  apply clarify apply (rule conjI) apply force
    apply (simp add: no-dyn-lookup)
  apply clarify apply (rule conjI) apply force
    apply (simp add: no-dyn-lookup)
    apply (case-tac c) apply simp apply simp apply simp apply simp
  apply clarify apply (rule conjI)
    apply clarify apply (erule impE) apply (simp add: no-dyn-fun)
    apply (case-tac no-dyn τ1) apply simp apply simp
    apply clarify
  apply (erule impE) apply simp apply (case-tac no-dyn τ1)
    apply simp apply simp apply simp apply (case-tac no-dyn τ1) apply simp apply simp
  apply clarify
proof −
  fix S Γ n0 n1 e τ1 n2 e′ τ2
  assume st12b: S ⊢ τ1 ≃ τ2 → UVarT n2 and ndg: no-dyn Γ
    and nda: no-dyn (App e e′)
    and IH1: no-dyn Γ ∧ no-dyn e ⟶ S;Γ ⊢ e : τ1 ∧ no-dyn τ1
    and IH2: no-dyn Γ ∧ no-dyn e′ ⟶ S;Γ ⊢ e′ : τ2 ∧ no-dyn τ2
  from ndg nda IH1 have wte: S;Γ ⊢ e : τ1 by auto
  from ndg nda IH1 have ndt1: no-dyn τ1 by auto
  from ndg nda IH2 have wtep: S;Γ ⊢ e′ : τ2 apply auto apply (case-tac no-dyn e)
    apply simp apply simp done
  from ndg nda IH2 have ndt2: no-dyn τ2 apply auto apply (case-tac no-dyn e)
    apply simp apply simp done
  from ndt2 have ndt2b: no-dyn (τ2 → UVarT n2) by simp
```

```
  from st12b ndt1 ndt2b have st1t2b: $S τ1 = $S (τ2 → UVarT n2)
    by (rule conseq-no-dyn-equal)
  from wte wtep st1t2b show S;Γ ⊢ App e e′ : UVarT n2 ∧ no-dyn (UVarT n2)
    by force
qed
```

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] C. Anderson and S. Drossopoulou. BabyJ - from object based to class based programming via types. In *WOOD '03*, volume 82, pages 53–81. Elsevier, 2003.

[3] R. Cartwright and M. Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, New York, NY, USA, 1991. ACM Press.

[4] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics*, 1995.

[5] C. Flanagan. Hybrid type checking. In *POPL 2006: The 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256, Charleston, South Carolina, January 2006.

[6] C. Flanagan and M. Felleisen. Componential set-based analysis. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 235–248, New York, NY, USA, 1997. ACM Press.

[7] C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Trans. Program. Lang. Syst.*, 21(2):370–416, 1999.

[8] D. P. Friedman, C. T. Haynes, and M. Wand. *Essentials of programming languages (3rd ed.).* MIT Press, Cambridge, MA, USA, 2008.

[9] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In R. Findler, editor, *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

[10] J. J. Heiss. Meet Peter von der Ahé, tech lead for Javac at Sun Microsystems. *Sun Developer Network (SDN)*, April 2007.

[11] D. Herman and C. Flanagan. Status report: specifying JavaScript with ML. In *ML '07: Proceedings of the 2007 workshop on Workshop on ML*, pages 47–52, New York, NY, USA, 2007. ACM.

[12] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *Trends in Functional Prog. (TFP)*, page XXVIII, April 2007.

[13] G. Huet. *Resolution d'equations dans les langages d'ordre 1, 2, ..., omega.* PhD thesis, Université Paris VII, France, 1976.

[14] G. L. S. Jr. An overview of COMMON LISP. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 98–107, New York, NY, USA, 1982. ACM Press.

[15] K. Knight. Unification: a multidisciplinary survey. *ACM Comput. Surv.*, 21(1):93–124, 1989.

[16] X. Leroy. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon.

[17] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[18] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[19] W. Naraschewski and T. Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *J. Autom. Reason.*, 23(3):299–318, 1999.

[20] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[21] M. S. Paterson and M. N. Wegman. Linear unification. In *STOC '76: Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 181–186, New York, NY, USA, 1976. ACM Press.

[22] S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*, December 2002.

[23] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[24] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[25] F. Pottier. A framework for type inference with subtyping. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 228–238, New York, NY, USA, 1998. ACM Press.

[26] F. Pottier. Simplifying subtyping constraints: a theory. *Inf. Comput.*, 170(2):153–183, 2001.

[27] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[28] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[29] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

[30] J. G. Siek and W. Taha. Gradual typing for objects. In *ECOOP 2007*, volume 4609 of *LCNS*, pages 2–27. Springer Verlag, August 2007.

[31] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[32] S. Thatte. Quasi-static typing. In *POPL 1990*, pages 367–381, New York, NY, USA, 1990. ACM Press.

[33] S. Tobin-Hochstadt. The design and implementatino of typed scheme. In *POPL 2008: The 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2008.

[34] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *OOPSLA'06 Companion*, pages 964–974, NY, 2006. ACM.

[35] P. Wadler. Theorems for free! In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, New York, NY, USA, 1989. ACM.

[36] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In D. Dube, editor, *Workshop on Scheme and Functional Programming*, pages 15–26, 2007.

[37] B. Wagner. Local type inference, anonymous types, and var. *Microsoft Developer Network*, 2007.

[38] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informatica*, 10:115–122, 1987.