A Functional View of Join

Martin Odersky, Christoph Zenger, Matthias Zenger
École Polytechnique Fédérale de Lausanne

Gang Chen
University of South Australia

October 14, 1999

Abstract

Join calculus, usually presented as a process calculus, is suitable as a foundation of both sequential and concurrent programming. We give a new operational semantics of join calculus, expressed as a reduction system with a single reduction rule similar to β reduction in lambda calculus. We also introduce a new Hindley/Milner style type system for join calculus. Compared to previous work, the type system gives more accurate types of composite and mutually recursive definitions. The type system's soundness is established by showing that our reduction rule keeps typings invariant. We present an algorithm for type inference and show its soundness and completeness.

1 Introduction

Foundational work on programming has been based traditionally on some variant of lambda calculus [Chu51], e.g. [Plo75, Fel88, Mog89, CF91, ORH93, AFM+95]. These calculi, ideally suited to sequential programming, are increasingly at odds with modern programs which are reactive in their interfaces and concurrent in their implementation. Process calculi such as π calculus [MPW92, Bou92, HY93] or fusion calculus [PV98] express the interaction aspects of computation well but lack abstraction when it comes to encoding functional and sequential computation. Laws of sequential programming usually can be recovered only with the aid of sophisticated type systems [PS93, KPT96] or name sortings [Ode95b].

Join calculus [FGL+96, FG96] can provide a unifying basis of call-by-value functional programming, sequential imperative programming and concurrent programming, and at the same time is simpler than previous calculi which try to model some combination of these areas [Bou89, Ode95a, Bou97]. As such it seems well suited to take over from lambda calculus as a generally agreed-upon foundation of programming. Join calculus has traditionally been introduced as a process calculus, and its operational semantics has been explained as a chemical abstract machine [BB90, FG96]. We show in this paper that join calculus can be regarded equivalently as a concurrent version of lambda calculus, with an operational semantics expressed as a reduction system. Our presentation of join calculus has a single reduction rule, similar to β_V-reduction in the call-by-value lambda calculus.

The motivations for this new view of join calculus are twofold. First, it strengthens the ties to lambda calculus and functional programming, which should help in carrying over results established in these areas. Second, it arguably brings with it important simplifications in the presentation of the calculus, which helps in explaining its concepts better and in streamlining proofs of technical results. For instance, the proof of our subject reduction theorem could be simplified considerably by basing it on a reduction system instead of a chemical abstract machine.

The second contribution of this paper is a revised and strengthened Hindley/Milner style type system for join calculus. Implicit polymorphism, common in functional languages, has been difficult to adapt to process calculi [Tur93]. Join calculus is an exception. Indeed, Fournet et al. [FLMR97] have proposed a Hindley/Milner style type system which provides a sound typing of joins. However, their type system still lacks polymorphism when compared to standard practice in functional programming. For instance, in a mutually recursive definition such as

```
even(xs) = isEmpty(xs)  || odd(tail(xs)),
odd(xs)  = ~isEmpty(xs) && even(tail(xs))
```

their type system would force `even` and `odd` to have monomorphic type. By contrast, in most typed functional programming languages such a composite definition would be encoded by the type checker as a fixed point operation in a let, and `even` and `odd` could both have the polymorphic type $\forall\alpha.\mathtt{List}\ \alpha \rightarrow \mathtt{Boolean}$. The functional view of mutual recursion as a fixed point operation is incompatible with joins on the left-hand sides of definitions. However, we show how the polymorphism present in functional programs can be recovered by a new typing rule for local definitions.

The rest of this summary is structured as follows. Section 2 explains join calculus informally, by means of small example programs. Section 3 introduces the untyped join calculus as a reduction system. Section 4 presents our type system. Section 5 shows that the type system is semantically sound. Section 6 presents a type inference algorithm and shows that it is sound and complete. Section 7 concludes.

2 Background

Join calculus provides a simple kernel language in which call-by-value functional programming, sequential imperative programming, and concurrent programming can all be expressed. It can be characterized as a name-passing lambda calculus, extended with parallel composition and a simple construct for the synchronization of parallel computations. Here's an example of a one place buffer written in join calculus:

```
put(k, x) & empty()  =  k() & full(x),
get(k) & full(x)     =  k(x) & empty()
```

Two mutually recursive equations define four functions, `put`, `get`, `empty` and `full`. The `&` operator represents a *fork* (or parallel composition) on the right hand side of an equation, and a *join* on the left-hand side. Join calculus programs can be given a simple reduction semantics. In the example above, if a call to `get(l)` and a call to `full(y)` are present, then the two calls are rewritten to the term `l(y) & empty()`. A single call to either `get` or `full` would block until its partner is present.

Note that this is similar to the execution of Petri nets [Rei85], with function calls corresponding to places and equations corresponding to transitions. In both systems, a transition (equation) fires (rewrites) if all incoming places have tokens (all left-hand side calls are present). But join calculus programs have considerably more power than Petri nets since calls can carry arguments and definitions can be nested.

Note also that the example above uses continuation passing style. Instead of returning a result directly, functions pass their result to a continuation argument, usually named `k`. Basic join calculus enforces continuation passing style, but it is possible to define direct style extensions, which are mapped to basic join calculus via a continuation passing transform.

If we leave out the `&` operator, join calculus can be seen as a minimal functional language. For instance, here is the definition of function composition in join calculus:

```
compose(f, g, k)  =  def
                         h(x, k2) = def
                                      k1(y) = f(y, k2)
                                  in  g(x, k1)
                     in  k(h)
```

If we don't insist on continuation passing style, the function can be written in a more concise and familiar way as follows:

```
compose(f, g)  =  def h(x) = f(g(x)) in h
```

Finally, join calculus can also be seen as a basis for imperative programming, since a mutable reference can be encoded by the following two definitions of a reader function `get` and a writer function `put`, coupled via an internal function `state`.

```
get(k) & state(x)     =  k(x) & state(x),
put(y, k) & state(x)  =  k() & state(y)
```

In [FLMR97] Fournet et al. have shown that the close correspondence of join calculus with functional languages extends to type systems. They have adapted the popular Hindley/Milner system [Mil78] to join calculus, providing a sound typing of effects. Their idea is that in a definition where several functions are defined only type variables that appear in the type of a single function can be generalized. This restriction prevents type variables that appear in the types of two or more functions in a join pattern from being generalized. Such a generalization would be unsound since it would break the coupling of the types of the co-defined functions. Their approach provides a sound typing of references via their encoding into join calculus. No special treatment like weak type variables or effect types is necessary. Note however that join calculus provides a trivial form of value polymorphism [Wri95], since defined names are always functions.

For instance, in the example above the most general types for `get`, `state` and `put` are:

$$\begin{array}{lcl} \texttt{get} & : & (\alpha \rightarrow \diamond) \rightarrow \diamond \\ \texttt{put} & : & (\alpha, () \rightarrow \diamond) \rightarrow \diamond \\ \texttt{state} & : & \alpha \rightarrow \diamond \end{array}$$

Here, $\tau \rightarrow \diamond$ is the type of a function with parameter τ which does not return a result. None of these functions is polymorphic since each function shares its type variable α with all others.

This is all as it should be. However, the Fournet et al. system suffers from an annoying shortcoming: Functions whose types share type variables because one function calls the other are treated in the same way as participants of join patterns, and therefore cannot be polymorphic. For instance, in

```
f(x, k) = g(x, k),
g(x, k) = k(x)
```

neither `f` nor `g` could be given their polymorphic ML-type $\forall\alpha.(\alpha, \alpha \rightarrow \diamond) \rightarrow \diamond$. One could imagine that the problem can be side-stepped by rearranging definitions after a dependency analysis. In our example, the definitions of `g` and `f` could each in turn be checked and generalized individually, thereby eliminating the unwanted coupling of type variables. But this approach breaks down when `f` and `g` are mutually recursive, since in this case neither can be checked without the other. The usual ML encoding of mutual recursion via a fixed point combinator is not generally applicable either since such an encoding does not take join patterns into account. Fournet et al. thus provide a sound type system for join patterns at the expense of an overly monomorphic typing of composite definitions.

In the following sections, we present a type system of join calculus that allows mutually recursive functions to be polymorphic. Its main novelty is a new rule for local definitions which regards generalization of bindings as a fixed point operation. Soundness of the new type system is established by giving an untyped operational semantics with structural equivalence rules and a β reduction rule, and showing that types are invariant under both equivalence and reduction. We also present a type inference algorithm for our system and show its soundness and completeness.

3 The Untyped Core Calculus

In this section we present our version of the untyped join calculus. The main difference compared to the original treatment [FGL+96, FG96] is that we express the operational semantics as a reduction system rather than as a chemical abstract machine. This streamlines both the presentation and the proofs. The syntax of our core calculus is given as follows.

Names	$a, b, c, \ldots, x, y, z$		
Terms	M	$=$	$\mathbf{def}\, D\, \mathbf{in}\, M \mid x(\overline{y}) \mid M \,\&\, M$
Definitions	D	$=$	$L = M \mid D, D \mid \epsilon$
Left-hand sides	L	$=$	$x_1(\overline{y}_1) \,\&\, \ldots \,\&\, x_n(\overline{y}_n) \qquad (n \geq 1)$

Meta-Syntactical Convention. For any expression X, we use $\overline{X}$ to indicate a sequence $X_1, \ldots, X_n$ of X's, where $n \geq 0$.

As terms M we have function calls $x(\overline{y})$, parallel composition expressed using the & operator, and local definitions, **def** D **in** M. A definition D is a possibly empty set of rewrite rules $L = M$. The left hand side of a rewrite rule is a *join pattern* $x_1(\overline{y}_1) \,\&\, \ldots \,\&\, x_n(\overline{y}_n)$, which defines a sequence of function names with their arguments. We require join patterns to be *linear*. That is, all function symbols in a join pattern must be pairwise different, and all function arguments must also be pairwise different. The right-hand side of a rewrite rule is an arbitrary term. The rewrite rules in a definition can be directly and mutually recursive.

The original treatment of join calculus used (|) for parallel composition and joins. We have changed this to (&) since joins and parallel composition represent conjunctions of events, whereas (|) is associated with logical or.

Definition. The set of *defined names* $\mathrm{dn}(L)$ and the set of *local names* $\mathrm{ln}(L)$ of a left-hand side L are defined as follows.

$$\begin{array}{lcl} \mathrm{dn}(x_1(\overline{y}_1) \,\&\, \ldots \,\&\, x_n(\overline{y}_n)) & = & \{x_1, \ldots, x_n\} \\ \mathrm{ln}(x_1(\overline{y}_1) \,\&\, \ldots \,\&\, x_n(\overline{y}_n)) & = & \{\overline{y}_1\} \cup \ldots \cup \{\overline{y}_n\} \end{array}$$

The sets of local and defined names $\mathrm{ln}(D)$, $\mathrm{dn}(D)$ of a definition D are the union of the corresponding sets $\mathrm{ln}(L)$, $\mathrm{dn}(L)$ of all its left hand sides L. The set of *free names* $\mathrm{fn}(M)$, $\mathrm{fn}(D)$ of a term and a definition are recursively defined as follows.

$$\begin{array}{lcl@{\qquad}lcl} \mathrm{fn}(\mathbf{def}\, D\, \mathbf{in}\, M) & = & (\mathrm{fn}(D) \cup \mathrm{fn}(M)) \backslash \mathrm{dn}(D) & \mathrm{fn}(L = M) & = & \mathrm{fn}(M) \backslash \mathrm{ln}(L) \\ \mathrm{fn}(x(\overline{y})) & = & \{x, \overline{y}\} & \mathrm{fn}(D_1, D_2) & = & \mathrm{fn}(D_1) \cup \mathrm{fn}(D_2) \\ \mathrm{fn}(M_1 \,\&\, M_2) & = & \mathrm{fn}(M_1) \cup \mathrm{fn}(M_2) & \mathrm{fn}(\epsilon) & = & \emptyset \end{array}$$

All names occurring in a term M that are not free in M are called *bound* in M.

Hygiene condition. We assume that the set of free and bound names of every term we write are disjoint. This can always be achieved using a suitable α-renaming (see below).

Definition. A *renaming* is an idempotent map from names to names that is the identity except on a finite number of names. We let θ range over renamings. We write $\mathrm{dom}(\theta)$ for the set of names where θ is not the identity, and $\mathrm{codom}(\theta)$ for its image under θ.

Definition. *Structural equivalence* $\equiv$ is the smallest compatible equivalence relation between terms that satisfies the following laws.

1. α-renaming:
$$\begin{array}{lcll} \mathbf{def}\, D \,\mathbf{in}\, M & \equiv & \mathbf{def}\, \theta D \,\mathbf{in}\, M & \text{if } \mathrm{dom}(\theta) \subseteq \mathrm{ln}(D) \wedge \mathrm{codom}(\theta) \cap \mathrm{fn}(D) = \emptyset \\ \mathbf{def}\, D \,\mathbf{in}\, M & \equiv & \mathbf{def}\, \theta D \,\mathbf{in}\, \theta M & \text{if } \mathrm{dom}(\theta) \subseteq \mathrm{dn}(D) \wedge \mathrm{codom}(\theta) \cap \mathrm{fn}(D, M) = \emptyset. \end{array}$$
In both equations we assume that θ is one-to-one.

2. (&) on terms is associative and commutative:
$$\begin{array}{rcl} M_1 \,\&\, M_2 & \equiv & M_2 \,\&\, M_1 \\ M_1 \,\&\, (M_2 \,\&\, M_3) & \equiv & (M_1 \,\&\, M_2) \,\&\, M_3 \end{array}$$

3. $(,)$ on definitions is associative and commutative with ϵ as an identity:
$$\begin{array}{rcl} D_1, D_2 & \equiv & D_2, D_1 \\ D_1, (D_2, D_3) & \equiv & (D_1, D_2), D_3 \\ D, \epsilon & \equiv & D \end{array}$$

4. Scope extrusion:
$$(\mathbf{def}\, D \,\mathbf{in}\, M_1) \,\&\, M_2 \quad \equiv \quad \mathbf{def}\, D \,\mathbf{in}\, (M_1 \,\&\, M_2) \qquad \text{if } \mathrm{dn}(D) \cap \mathrm{fn}(M_2) = \emptyset$$

In the following, we will always identify terms that are structurally equivalent.

Definition. *Reduction* $\rightarrow$ is the smallest compatible relation that satisfies the β-reduction law:

$$\mathbf{def}\, D, L = M \,\mathbf{in}\, R[\theta L] \quad \rightarrow \quad \mathbf{def}\, D, L = M \,\mathbf{in}\, R[\theta M] \qquad \text{if } \mathrm{dom}(\theta) \subseteq \mathrm{ln}(L)$$

Here R is a *reduction context* that determines where reductions can take place. The set of all reduction contexts is given by the following grammar:

$$R \quad = \quad [\,] \mid \mathbf{def}\, D \,\mathbf{in}\, R \mid R \,\&\, M \mid M \,\&\, R$$

The expression $R[M]$ denotes the reduction context R with its hole filled with M.

The possible shapes of reduction contexts determine which calls can be "executed", replacing some calls by the right-hand side of their corresponding definition. Generally, all calls can be executed except those that are part of the right hand side of some local definition. The definition of allowable redexes via a reduction context is common in sequential programming, however, the term *evaluation context* is usually employed. We have changed that name since there is nothing to evaluate in continuation passing style join calculus.

The operational semantics given here corresponds closely to the original treatment [FGL+96]. Where we use term reduction of $\mathbf{def}\ D, L = M\ \mathbf{in}\ R[\theta L]$, they use reduction in a chemical abstract machine with a "molecule soup" that contains L and an environment that contains the definitions D, $L = M$ and all definitions in R. We don't need their heating and cooling rules that transfer definitions from the soup to the environment and back. Our structural equivalence is also similar to theirs, except that we don't have a law that fuses several definitions (i.e. their S-AND rule would correspond to an equivalence **def** D **in def** D' **in** $M \equiv$ **def** D, D' **in** M which is absent in our setting). This rule, which simplifies their chemical abstract machine reduction, is not necessary for our term reduction. Notably the fusion rule also breaks subject reduction; for this reason the original typed treatment of join calculus [FLMR97] abandons the law and introduces structured environments.

4 Type System

The syntax of types, type schemes, and environments is given as follows.

Type variables	α, β, γ		
Types	τ	$=$	$\alpha \mid (\tau_1, \ldots, \tau_n) \rightarrow \diamond$
Type schemes	σ	$=$	$\tau \mid \forall\alpha.\sigma$
Environments	Δ	$=$	$x_1 : \tau_1, \ldots, x_n : \tau_n$
	Γ	$=$	$x_1 : \sigma_1, \ldots, x_n : \sigma_n$

Types are either type variables or function types. The result part of a function type is always $\diamond$ which stands for "undefined". As in the Hindley/Milner system we distinguish type schemes from types. Unlike types, type schemes can be polymorphic; that is they can contain universally quantified type variables. Environments Γ bind names to type schemes. We distinguish monomorphic environments Δ as a subset of environments which bind all names to types.

We will need several functions and operators on environments. The *domain* of an environment $\mathrm{dom}(\Gamma)$ is the set of identifiers defined in the environment. The set of type variables occurring free in some type in Γ is written $\mathrm{tv}(\Gamma)$. The $(,)$ operator on environments is associative, with the empty environment as left and right identity. The operands Γ_1 and Γ_2 of a composite environment (Γ_1, Γ_2) are required to have disjoint domains. Proof trees where this requirement is violated are not considered valid. $\Gamma_1 \oplus \Gamma_2$ denotes the concatenation of the environments Γ_1 and Γ_2, where a binding in Γ_2 replaces any binding of the same name in Γ_1. $\Gamma_1 \cup \Gamma_2$ denotes map union of the two environments Γ_1 and Γ_2, where it is required that names bound in both environments are bound to the same type.

There are three forms for *typing judgements*, which apply to identifiers, definitions, and terms:

$\Gamma \vdash x : \sigma$	The identifier x has type scheme σ.
$\Gamma \vdash L = M : \Delta$	The rewrite rule $L = M$ produces environment Δ.
$\Gamma \vdash M : \diamond$	The term M is well-typed.

A typing judgement is *valid* if it is derivable by applications of the typing rules given below.

$\boxed{\Gamma \vdash x : \sigma}$

$$(\textsc{Taut})\ \ \Gamma \vdash x : \sigma \qquad (x : \sigma \in \Gamma) \qquad\qquad (\forall\mathrm{E})\ \frac{\Gamma \vdash x : \forall\alpha.\sigma}{\Gamma \vdash x : [\tau/\alpha].\sigma}$$

$\boxed{\Gamma \vdash L = M : \Delta}$

$$(\textsc{Eqn})\ \frac{\Gamma \oplus (\overline{y}_1 : \overline{\tau}_1, \ldots, \overline{y}_n : \overline{\tau}_n) \vdash M : \diamond}{\Gamma \vdash x_1(\overline{y}_1)\ \&\ \ldots\ \&\ x_n(\overline{y}_n) = M : (x_1 : (\overline{\tau}_1) \rightarrow \diamond, \ldots, x_n : (\overline{\tau}_n) \rightarrow \diamond)}$$

$\boxed{\Gamma \vdash M : \diamond}$

$$(\text{App})\ \frac{\Gamma \vdash x : (\tau_1, \ldots, \tau_n) \rightarrow \diamond \qquad \Gamma \vdash y_1 : \tau_1 \quad \ldots \quad \Gamma \vdash y_n : \tau_n}{\Gamma \vdash x(y_1, \ldots, y_n) : \diamond}$$

$$(\text{Par})\ \frac{\Gamma \vdash M_1 : \diamond \qquad \Gamma \vdash M_2 : \diamond}{\Gamma \vdash M_1 \,\&\, M_2 : \diamond}$$

$$(\text{Def})\ \frac{\begin{array}{c}\Gamma \oplus (\Delta_1 \cup \ldots \cup \Delta_n) \vdash D_i : \Delta_i \qquad \Gamma \oplus (\Gamma'_1 \cup \ldots \cup \Gamma'_n) \vdash \Delta_i \ \mathbf{gen}\ \Gamma'_i \quad (i = 1, \ldots, n) \\ \Gamma \oplus (\Gamma'_1 \cup \ldots \cup \Gamma'_n) \vdash M : \diamond\end{array}}{\Gamma \vdash \mathbf{def}\, D_1, \ldots, D_n \,\mathbf{in}\, M : \diamond}$$

Rule (Def) uses an auxiliary judgement $\Gamma \vdash \Delta\ \mathbf{gen}\ \Gamma'$ which states that under the global environment Γ, the environment Δ may be generalized to Γ'. Deduction rules for this judgement are given below.

$$(\text{Refl})\ \Gamma \vdash \Delta\ \mathbf{gen}\ \Delta \qquad\qquad (\forall\text{I})\ \frac{\Gamma \vdash \Delta\ \mathbf{gen}\ (x : \sigma, \Gamma')}{\Gamma \vdash \Delta\ \mathbf{gen}\ (x : \forall\alpha.\sigma, \Gamma')}\ (\alpha \notin \mathrm{tv}(\Gamma, \Gamma'))$$

These rules are the equivalent of the ($\forall$I) rule of the Hindley/Milner system. They embody the restriction of join calculus that type variables which appear in the type of more than one function of a left hand side may not be generalized.

The (Def) rule embodies the main innovation of the type system we present. It uses a double fixed point construction to deal with possible recursion between rewrite rules D_i. First, each rewrite rule D_i is checked under the monomorphic environment created by all definitions together. Then each monomorphic environment is generalized under the polymorphic environment created by all generalizations together.

It might appear more natural to check each definition immediately under the polymorphic environment created by all definitions and generalizations. Such a rule can express polymorphic recursion, which also means that it does not admit complete type inference [Hen93]. Rule (Def) splits the single fixed point of the polymorphic recursion rule into two successive fixed points, which can both be determined by type inference.

Examples We illustrate the expressiveness of the type system with a number of examples. Consider first the definition of a local channel:

```
receive(k) & send(x)  =  k(x)
```

This will produce the environment ($\texttt{receive} : (\alpha \rightarrow \diamond) \rightarrow \diamond, \texttt{send} : \alpha \rightarrow \diamond$). The type variable α cannot be generalized since it appears in the types of both `receive` and `send`.

Next, consider the definition of a channel creator:

```
newChannel(k')  =  def receive(k) & send(x) = k(x)
                   in  k'(receive, send)
```

Then `newChannel` has type $\forall\alpha.((\alpha \rightarrow \diamond) \rightarrow \diamond, \alpha \rightarrow \diamond) \rightarrow \diamond) \rightarrow \diamond$. The type variable α can be generalized, even though it couples the types of `receive` and `send`. But both `receive` and `send` are local and thus do not form part of the generated environment

$$\texttt{newChannel} : ((\alpha \rightarrow \diamond) \rightarrow \diamond, \alpha \rightarrow \diamond) \rightarrow \diamond) \rightarrow \diamond\ .$$

In this environment, α appears only in the binding for `newChannel`, i.e. the type variable is no longer shared between several bindings. Therefore, the binding may be generalized.

Finally, consider the following definitions:

```
f(x, k) & tick()  =  g(x, k)
g(x, k) & tick()  =  f(x, k) & k(x)
```

The (admittedly a little contrived) purpose of this program fragement is to wait for an initial call of `f(x,k)`, after which the continuation `k` is called with the provided argument `x` on every second call to `tick`. The intermediate monomorphic environment Δ produced by rule (Def) is

$$\begin{array}{lcl} \texttt{f} & : & (\alpha, \alpha \to \diamond) \to \diamond \\ \texttt{g} & : & (\alpha, \alpha \to \diamond) \to \diamond \\ \texttt{tick} & : & () \to \diamond \end{array}$$

This is also the final environment produced by the original join calculus type system [FLMR97]. By contrast, in our type system the types of `f` and `g` can both be generalized to $\forall \alpha.(\alpha, \alpha \to \diamond) \to \diamond$.

5 Type Soundness

In this section, we sketch the proofs of two theorems which state that typings are invariant under structural equivalence and reduction.

Theorem 5.1 If $M \equiv N$ and $\Gamma \vdash M : \diamond$ then $\Gamma \vdash N : \diamond$.

Theorem 5.2 (Subject Reduction) If $M \to N$ and $\Gamma \vdash M : \diamond$ then $\Gamma \vdash N : \diamond$.

Subject reduction is as far as we can get in characterizing type soundness for join calculus. It says that any reduction from a well-typed term gives a sequence of well-typed terms. For sequential languages, one usually adds the requirement that every typable term which cannot be further reduced represents an answer. In a concurrent language this requirement would imply that deadlocks are impossible for typed programs. But our type system is not strong enough to guarantee absence of deadlocks. Nevertheles, the type system is strong enough to guarantee the absence of certain kinds of run-time errors. For example, one can guarantee that function calls with a wrong number of arguments are impossible.

To prove invariance of typings under structural equivalence and reduction, we need two preliminary lemmas, which can both be shown by standard techniques.

Lemma 5.3 (Context) For all environments Γ:

$$\forall C.\, \exists \Gamma'.\, \forall M. \quad \Gamma \vdash C[M] : \diamond \quad \Longleftrightarrow \quad \Gamma, \Gamma' \vdash M : \diamond$$

Lemma 5.4 (Weakening) For all environments Γ, Γ' and terms M:

$$\Gamma \vdash M : \diamond \quad \Rightarrow \quad \Gamma, \Gamma' \vdash M : \diamond$$

provided (Γ, Γ') is well-defined, i.e. $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \emptyset$.

To show Theorem 5.1, one shows that all laws of structural invariance maintain typability. This is non-trivial only for the scope extrusion law, which requires an application of the weakening lemma. One then appeals to the context lemma to show that the invariance also applies to the compatible closure of $(\equiv)$.

The key step in the proof of subject reduction (Theorem 5.2) is to show that

$$\Gamma \vdash \textbf{def}\, D, L = M \textbf{ in } R[\theta L] : \diamond \ \Rightarrow \ \Gamma \vdash \textbf{def}\, D, L = M \textbf{ in } R[\theta M] : \diamond \tag{1}$$

To show implication (1), we introduce two auxiliary judgements that associate definitions and contexts with environments. The first judgement, of the form $\Gamma \vdash D :_{\text{gen}} \Gamma'$, associates a definition with the environment it is given in rule (DEF'):

$$(\text{DEF'})\ \frac{\Gamma \oplus (\Delta_1 \cup \ldots \cup \Delta_n) \vdash D_i : \Delta_i \qquad \Gamma \oplus (\Gamma'_1 \cup \ldots \cup \Gamma'_n) \vdash \Delta_i \textbf{ gen } \Gamma'_i \quad (i = 1, \ldots, n)}{\Gamma \vdash (D_1, \ldots, D_n) :_{\text{gen}} (\Gamma'_1 \cup \ldots \cup \Gamma'_n)}$$

The second judgement, of the form $\Gamma \vdash R :_{\text{gen}} \Gamma_R$, associates a reduction context with the environment represented by all definitions which are visible at the hole of the context. This judgement is derived from the following rules.

$$(R_{[\,]})\ \Gamma \vdash [\,] :_{\text{gen}} \epsilon \qquad\qquad (R_{\&})\ \frac{\Gamma \vdash R :_{\text{gen}} \Gamma' \qquad \Gamma \vdash M : \diamond}{\Gamma \vdash R \& M :_{\text{gen}} \Gamma'}$$

$$(R_{\textbf{in}})\ \frac{\Gamma \vdash D :_{\text{gen}} \Gamma' \qquad \Gamma, \Gamma' \vdash R :_{\text{gen}} \Gamma''}{\Gamma \vdash \textbf{def}\, D \textbf{ in } R :_{\text{gen}} \Gamma', \Gamma''}$$

Observe that environment formation with $(,)$ is always well-defined since we assume terms to be hygienic, so there is no overlap in the environments' domain.

The key property of the above judgements is:

Lemma 5.5

$$\begin{array}{l}\Gamma \vdash R[M] : \diamond \Rightarrow \exists \Gamma_R.\ \Gamma \vdash R :_{\text{gen}} \Gamma_R \ \wedge\ \Gamma, \Gamma_R \vdash M : \diamond \\ \Gamma \vdash R :_{\text{gen}} \Gamma_R \ \wedge\ \Gamma, \Gamma_R \vdash M : \diamond \Rightarrow \Gamma \vdash R[M] : \diamond\end{array}$$

Now, to show implication (1), assume its left-hand side is valid and pick an environment Γ_D such that $\Gamma \vdash (D, L = M) :_{\text{gen}} \Gamma_D$ and $\Gamma, \Gamma_D \vdash R[\theta L] : \diamond$. An application of Lemma 5.5 with $R \equiv \textbf{def}\ D, L = M \textbf{ in } [\,]$ guarantees that such an environment exists. Next, pick an environment Γ_R such that $\Gamma, \Gamma_D \vdash R :_{\text{gen}} \Gamma_R$ and $\Gamma, \Gamma_D, \Gamma_R \vdash \theta L : \diamond$. Again Lemma 5.5 guarantees that such an environment exists. Next, we show that θM is typable under the same environment, that is, $\Gamma, \Gamma_D, \Gamma_R \vdash \theta M : \diamond$. Finally, we obtain $\Gamma \vdash \textbf{def}\, D, L = M \textbf{ in } R[\theta M] : \diamond$ by reference to the second implication in Lemma 5.5. With implication (1) established, Theorem 5.2 is now shown by a simple application of the context lemma (Lemma 5.3).

The complete and detailed proofs can be found in appendix 9.

6 Type Inference

In this section we present a type inference algorithm for the type system introduced before. The basic idea of type inference is adapted from the Hindley/Milner type system with the main difference lying in the rule for local definitions. Unlike in the Hindley/Milner system, definitions in Join can be composite and recursive and their left-hand sides can co-define several functions. To check a composite definition, we examine first the left-hand sides of all its equations, obtaining a monomorphic environment for every defined name. The right-hand sides are typed under this monomorphic environment. We then generalize the monomorphic environment to a possibly polymorphic one. Generalization can apply to all type variables that are not free in the enclosing environment, except for those that are present in the types of several co-defined functions. Soundness and completeness results for our type inference algorithm are given at the end of this section.

Before presenting the formalization of the type inference algorithm, we introduce some auxiliary functions in Figure 1. $inst(\sigma)$ replaces the bound type variables in type scheme σ with fresh type variables. Function *denv* takes a left-hand side of an equation and returns a new type environment for all the defined names. With function *lenv* it is possible to map such a type environment for a left-hand side $x_1(\overline{y}_1)\,\&\ldots\&\,x_n(\overline{y}_n)$ to the corresponding local type environment with domain $\{\overline{y}_1, \ldots, \overline{y}_n\}$. Finally, the last two functions *mtv* and *gen* are used for the generalization of monomorphic type environments. For a set of names, *mtv* returns all type variables that appear free in more than one of the names types. $gen(\Gamma, mono, \Delta)$ generalizes a complete type environment Δ by universally quantifying all free variables in types, that are not free in the environment Γ and that are not element of the set *mono*.

$$
\begin{array}{lcl}
inst(\forall\alpha_1\ldots\alpha_n.\tau) & = & \texttt{let}\quad \texttt{fresh}\ \beta_1, \ldots, \beta_n \\
 & & \texttt{in}\quad [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]\tau \\
denv(x_1(\overline{y}_1)\,\&\ldots\&\,x_n(\overline{y}_n)) & = & \texttt{let}\quad \texttt{fresh}\ \overline{\alpha}_1, \ldots, \overline{\alpha}_n \quad (|\overline{y}_i| = |\overline{\alpha}_i|) \\
 & & \texttt{in}\quad [x_1 : \overline{\alpha}_1 \to \diamond, \ldots, x_n : \overline{\alpha}_n \to \diamond] \\
lenv([x_1 : \overline{\tau}_1 \to \diamond, \ldots, x_n : \overline{\tau}_n \to \diamond], & & \\
\qquad x_1(\overline{y}_1)\,\&\ldots\&\,x_n(\overline{y}_n)) & = & [\overline{y}_1 : \overline{\tau}_1, \ldots, \overline{y}_n : \overline{\tau}_n] \\
mtv(\Delta, V) & = & \bigcup_{x,y\in V, x\neq y} \mathrm{tv}(\Delta(x)) \cap \mathrm{tv}(\Delta(y)) \\
gen(\Gamma,\, mono,\, [x_1 : \tau_1, \ldots, x_n : \tau_n]) & = & [x_1 : gen(\Gamma, mono, \tau_1), \ldots, x_n : gen(\Gamma, mono, \tau_n)] \\
gen(\Gamma,\, mono,\, \tau) & = & \texttt{let}\quad \{\alpha_1, \ldots, \alpha_n\} = \mathrm{tv}(\tau)\backslash(\mathrm{tv}(\Gamma) \cup mono) \\
 & & \texttt{in}\quad \forall\alpha_1\ldots\alpha_n.\tau
\end{array}
$$

Figure 1: Auxiliary functions for type inference

The inference algorithm is shown in Figure 2. It is an adaptation of algorithm W introduced by Damas and Milner in [DM82]. Like the former it reduces a type inference problem to a unification problem, which can be solved by a standard algorithm *mgu*. For different syntactic domains of our core calculus we define separate functions W_x, W_M and W_D.

Type inference for names and terms is straightforward, so we focus our interest on W_D. Given a substitution ϕ and a type environment Γ, W_D returns a polymorphic type environment Γ' and a new substitution ϕ' for a tuple of definitions $D_1, \ldots, D_n$. For every equation D_i, W_D first determines environments Δ_i consisting of all defined names. Then these environments are combined into a monomorphic type environment Δ by unifying types of names that are defined in more than one Δ_i. Now, for all equations $L_i = M_i$ type inference is applied to the right-hand sides M_i under a combination of the enclosing type environment Γ, environment Δ and the type environment local to the definition D_i. Finally, the types in Δ have to be generalized. As we mentioned above, type variables cannot be generalized if they appear free in the types of several co-defined names. Therefore we have to compute the set of non generalizable variables *mono* by applying function *mtv* to all equations. Every type τ in Δ can be generalized by universally quantifying all free type variables in τ that are neither free in the enclosing environment nor element of set *mono*.

It is obvious that the type inference algorithm terminates, since the expression argument is getting smaller in each recursive call. Furthermore W is sound and complete in the following sense:

Corollary 6.1 (Soundness) $W_M(id, \Gamma, M) = \phi \;\Rightarrow\; \phi\Gamma \vdash M : \diamond$.

Corollary 6.2 (Completeness) $\psi\Gamma \vdash M : \diamond \;\Rightarrow\; W_M(id, \Gamma, M) = \phi$ and $\phi \leq_{\mathrm{tv}(\Gamma)} \psi$.

$$
\begin{array}{lcl}
W_x & : & \mathit{Sub} \times \mathit{Env} \times \mathit{Name} \to \mathit{Type} \\
W_x(\phi, \Gamma,\, x) & = & \phi\; \mathit{inst}(\Gamma(x)) \\[1ex]
W_M & : & \mathit{Sub} \times \mathit{Env} \times \mathit{Term} \to \mathit{Sub} \\
W_M(\phi, \Gamma,\, x\,(y_1,\, \ldots,\, y_n)\,) & = & \texttt{let}\quad \texttt{all}\ i = 1 \ldots n\text{:}\ \tau_i = W_x(\phi, \Gamma, y_i) \\
 & & \texttt{in}\quad\ \mathit{mgu}(W_x(\phi, \Gamma, x)\ =\ \overline{\tau} \to \diamond)\phi \\
W_M(\phi, \Gamma,\, M_1 \,\&\, M_2) & = & W_M(W_M(\phi, \Gamma, M_1),\, \Gamma,\, M_2) \\
W_M(\phi, \Gamma,\, \mathbf{def}\, D\, \mathbf{in}\, M) & = & \texttt{let}\quad (\phi', \Gamma') = W_D(\phi, \Gamma,\, D) \\
 & & \texttt{in}\quad\ W_M(\phi',\, \Gamma \oplus \Gamma',\, M) \\[1ex]
W_D & : & \mathit{Sub} \times \mathit{Env} \times \mathit{Defn} \to \mathit{Sub} \times \mathit{Env} \\
W_D(\phi, \Gamma,\, L_1{=}M_1,\, \ldots,\, L_n{=}M_n) & = & \texttt{let}\quad \texttt{all}\ i = 1 \ldots n\text{:}\ \Delta_i = \mathit{denv}(L_i) \\
 & & \qquad\ \phi_0 = \mathit{mgu}(\bigwedge_{x \in \mathrm{dom}(\Delta_i) \cap \mathrm{dom}(\Delta_j)} \Delta_i(x) = \Delta_j(x))\phi \\
 & & \qquad\ \Delta = \phi_0 \Delta_1 \,\cup \ldots \cup\, \phi_0 \Delta_n \\
 & & \qquad\ \texttt{all}\ i = 1 \ldots n\text{:} \\
 & & \qquad\qquad \phi_i = W_M(\phi_{i-1},\, \Gamma \oplus \Delta \oplus \mathit{lenv}(\Delta_i, L_i),\, M_i) \\
 & & \qquad\ \mathit{mono}\ =\ \mathit{mtv}(\phi_n \Delta, \mathrm{dn}(L_1)) \cup \ldots \cup \mathit{mtv}(\phi_n \Delta,\, \mathrm{dn}(L_n)) \\
 & & \texttt{in}\quad\ (\phi_n,\, \mathit{gen}(\phi_n \Gamma,\, \mathit{mono},\, \phi_n \Delta))
\end{array}
$$

Figure 2: Type inference algorithm

All given and resulting substitutions are assumed/claimed to be idempotent and we write $\phi \leq_U^{\psi'} \psi$ iff $\psi'\phi = \psi'$, and $\psi'|_{U \cup \mathrm{dom}(\psi)} = \psi$ where $\mathrm{dom}(\phi) = \{u | \phi u \neq u\}$. This expresses the fact that ϕ is a more general substitution than ψ on U. We write $\phi \leq_U \psi$ if there exists a ψ' such that $\phi \leq_U^{\psi'} \psi$. Soundness includes that if M is not typable in an environment Γ, $W_M(\mathit{id}, \Gamma, M)$ will fail in some mgu-computation. A notion of principal types is hardly useful here, because the type of a term is always $\diamond$. The two above claims are special cases of these soundness and completeness theorems:

Theorem 6.3 (Soundness)

$$
\begin{array}{lcl}
W_x(\phi, \Gamma, x) = \tau & \Rightarrow & \phi\Gamma \ \vdash\ x : \tau \text{ and } \phi\tau = \tau \\
W_M(\phi, \Gamma, M) = \phi' & \Rightarrow & \phi'\Gamma \ \vdash\ M : \diamond \\
W_D(\phi, \Gamma, D) = (\phi', \Gamma') & \Rightarrow & \phi'\Gamma \ \vdash\ D\ :_{\mathrm{gen}}\ \Gamma' \text{ and } \phi'\Gamma' = \Gamma'
\end{array}
$$

and $\phi' = \phi'\phi$.

Theorem 6.4 (Completeness) If $\phi \leq_U^{\chi} \psi$ then

$$
\begin{array}{lcl}
\psi\Gamma \ \vdash\ x : \psi\tau & \Rightarrow & W_x(\phi, \Gamma, x) = (V, \tau') \text{ and } \chi'\tau' = \chi\tau \\
\psi\Gamma \ \vdash\ M : \diamond & \Rightarrow & W_M(\phi, \Gamma, M) = (V, \phi') \\
\psi\Gamma \ \vdash\ D\ :_{\mathrm{gen}}\ \psi\Gamma' & \Rightarrow & W_D(\phi, \Gamma, D) = (V, \phi', \Gamma'') \text{ and } \chi'\Gamma'' \subseteq \chi\Gamma'
\end{array}
$$

for some fresh V and $\phi' \leq_{C(V)}^{\chi'} \chi$, where $\phi' = \phi$ for the first case.

Here, a type scheme $\forall\overline{\beta}.\tau'$ is called a *generic instance* of $\forall\overline{\alpha}.\tau$ iff there exist $\overline{\tau}''$ with $[\overline{\tau}''/\overline{\alpha}]\tau = \tau'$ and $\mathrm{tv}(\tau) \cap \overline{\beta} = \emptyset$. We write $\forall\overline{\alpha}.\tau \subseteq \forall\overline{\beta}.\tau'$. An environment Γ' is a generic instance of Γ iff $\mathrm{dom}(\Gamma') \subseteq \mathrm{dom}(\Gamma)$ and for each $x \in \mathrm{dom}(\Gamma')$ we have $\Gamma(x) \subseteq \Gamma'(x)$. The relation $\subseteq$ is reflexive, transitive and stable under substitution. $C(V)$ denotes the complement of V with respect to the set of all type variables. We use the judgement $\Gamma \;\vdash\; D \;\;:_{\text{gen}}\;\; \Gamma'$, which was introduced in the section on type soundness.

The proofs of these theorems proceed on structural induction on the size of the term to type-check. The cases W_x and W_M are easily adapted from standard Hindley/Milner type systems and so is W_D but for the following generalization lemma, which reflects the new generalization rule.

Lemma 6.5 (Generalization)

$$\begin{array}{lcl} \forall i.\ \Gamma'_i = gen(\Gamma, \bigcup_j mtv(\Delta_j, \mathrm{dn}(L_j)), \Delta_i) & \Rightarrow & \forall i.\ \Gamma \oplus (\Gamma'_1 \cup \ldots \cup \Gamma'_n) \;\vdash\; \Delta_i \;\mathbf{gen}\; \Gamma'_i \\ \forall i.\ \Gamma \oplus (\Gamma'_1 \cup \ldots \cup \Gamma'_n) \;\vdash\; \Delta_i \;\mathbf{gen}\; \Gamma'_i & \Rightarrow & \forall i.\ gen(\Gamma, \bigcup_j mtv(\Delta_j, \mathrm{dn}(L_j)), \Delta_i) \subseteq \Gamma'_i \end{array}$$

For the first statement we check, that all variables, which are not in $\mathrm{tv}(\Gamma) \cup \bigcup_j mtv(\Delta_j, \mathrm{dn}(L_j))$ can really be generalized. The second statement is based on the observations that for arbitrary $\Gamma \;\vdash\; \Delta \;\;\mathbf{gen}\;\; \Gamma'$ we have $\mathrm{tv}(\Gamma) \cap \mathrm{tv}(\Delta) \subseteq \mathrm{tv}(\Gamma')$ and $\mathrm{tv}(\Delta(x_i)) \cap \mathrm{tv}(\Delta(x_j)) \subseteq \mathrm{tv}(\Gamma'(x_i)) \cap \mathrm{tv}(\Gamma'(x_j))$ for $i \neq j$.

7 Conclusion

We have re-interpreted join calculus as a reduction system, and have studied a Hindley/Milner style type system for it which is more accurate than previous work. This work was motivated by the desire to develop a simple foundation for languages that support functional, imperative, and concurrent programming. Current examples of such languages include JoCaml [Fes98], based on original join calculus, Pict [PT97] or Piccola [ALSN99], based on π calculus, as well as CML [Rep91], Facile [GMP89], or Erlang [AVW93], which are based on different, generally more complex foundations. We are currently defining and implementing a kernel programming language which is based on our interpretation of join calculus.

References

[AFM+95] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *Proc. 22nd ACM Symposium on Principles of Programming Languages*, pages 233–246, January 1995.

[ALSN99] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola - a small composition language. Submitted for Publication, available from `http://ww.iam.unibe.ch/~scg/Research/Piccola`, 1999.

[AVW93] J. Armstrong, R. Virding, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1993.

[BB90] Gérard Berry and Gérard Boudol. The chemical abstract machine. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, pages 81–94, January 1990.

[Bou89] Gérard Boudol. Towards a lambda-calculus for concurrent and communicating systems. In J. Díaz and F. Orejas, editors, *Proceedings TAPSOFT '1989*, pages 149–161, New York, March 1989. Springer-Verlag. Lecture Notes in Computer Science 351.

[Bou92] Gérard Boudol. Asynchrony and the pi-calculus. Research Report 1702, INRIA, May 1992.

[Bou97] Gérard Boudol. The pi-calculus in direct style. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 228–241, 1997.

[CF91] Erik Crank and Matthias Felleisen. Parameter-passing and the lambda-calculus. In *Proc. 18th ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 233–244, January 1991.

[Chu51] Alonzo Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematics Studies*. Princeton University Press, second edition, 1951.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, pages 207–211, January 1982.

[Fel88] Matthias Felleisen. λ-v-CS: An extended λ-calculus for scheme. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 72–84, Snowbird, Utah, July 1988.

[Fes98] Fabrice Le Fessant. *The JoCaml reference manual*. INRIA Rocquencourt, 1998. Available from `http://join.inria.fr`.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 21-24 1996. ACM.

[FGL$^+$96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421, Pisa, Italy, August 26-29 1996. Springer-Verlag. LNCS 1119.

[FLMR97] Cdric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Implicit typing à la ML for the join-calculus. In *Proc. of the 1997 8th International Conference on Concurrency Theory*. Springer-Verlag, 1997.

[GMP89] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(1):253–289, April 1993.

[HY93] Keiho Honda and Nobuko Yoshida. On reduction-based process semantics. In *Proc. 13th Conf. on Foundations of Softawre Technology and Theoretical Computer Science*, pages 373–387, December 1993.

[KPT96] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, 1996.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Dec 1978.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings 1989 IEEE Symposium on Logic in Computer Science*, pages 14–23. IEEE, June 1989.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.

[Ode95a] Martin Odersky. Applying π: Towards a basis for concurrent imperative programming. In *Proc. 2nd ACM SIGPLAN Workshop on State in Programming Languages*, January 1995.

[Ode95b] Martin Odersky. Polarized name passing. In *Proc. FST & TCS*, LNCS 1026, pages 324–337. Springer Verlag, December 1995.

[ORH93] Martin Odersky, Dan Rabin, and Paul Hudak. Call-by-name, assignment, and the lambda calculus. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 43–56, January 1993.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[PS93] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *IEEE Symposium on Logic in Computer Science*, June 1993.

[PT97] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press, 1999.

[PV98] Joachim Parrow and Björn Victor. The fusion calculus. In *Proc. 13th IEEE Symposium on Logic in Computer Science*, pages 176–185. IEEE Computer Society Press, 1998.

[Rei85] W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[Rep91] John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 293–305, June 1991.

[Tur93] David N. Turner. *The Pi calculus: Types, Polymorphism and Implementation.* PhD thesis, LFCS, University of Edinburgh, 1993.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356., December 1995.

8 Basic Properities

Starting from this section, we will give technical proofs for subject reduction and the correctness of type checking algorithm. This section contains lemmas which will be used in both proofs.

Without loss of generality, in the following, we assume that names defined in different definition block are disjoint (renaming bound names if necessary to avoid name clashes). Then the notation "$\oplus$" will be same as ",". We also use the notation $\Gamma \vdash \overline{y} : \overline{\tau}$ to denote the fact that $\Gamma \vdash y_i : \tau_i$ for $i = 1..n$. For a tuple $\overline{\alpha}$, the notation $|\overline{\alpha}|$ denotes the set of its members: $|\overline{\alpha}| = \{\alpha_1, \cdots, \alpha_n\}$.

Now we show that the typability is invariant under type substitution.

Lemma 8.1 $\quad \Gamma \vdash x : \tau_a \Rightarrow [\tau_b/\alpha]\Gamma \vdash x : [\tau_b/\alpha]\tau_a$

Proof

$$\begin{array}{llll}
\Gamma \vdash x : \tau_a & \Rightarrow & x : \forall\overline{\alpha'}.\tau_c \in \Gamma \ \wedge \ \tau_a = [\overline{\tau}/\overline{\alpha'}]\tau_c & \\
\Gamma \vdash x : \tau_a & \Rightarrow & x : \forall\overline{\alpha'}.[\tau_b/\alpha]\tau_c \in [\tau_b/\alpha]\Gamma & \\
 & \Rightarrow & [\tau_b/\alpha]\Gamma \vdash x : [[\tau_b/\alpha]\overline{\tau}/\overline{\alpha'}][\tau_b/\alpha]\tau_c & \\
 & \Rightarrow & [\tau_b/\alpha]\Gamma \vdash x : [\tau_b/\alpha][\overline{\tau}/\overline{\alpha'}]\tau_c & |\overline{\alpha'}| \cap tv(\tau_b) = \emptyset \\
 & \Rightarrow & [\tau_b/\alpha]\Gamma \vdash x : [\tau_b/\alpha]\tau_a &
\end{array}$$

□

Note that the rule $\forall I$ can be safely replaced by the following $\forall I'$ without changing the typing.

$$(\forall I') \ \frac{\Gamma \vdash \Delta \ \textbf{gen} \ (x : \sigma, \Gamma')}{\Gamma \vdash \Delta \ \textbf{gen} \ (x : \forall\alpha.\sigma, \Gamma')} \ (\alpha \in \mathrm{tv}(\Delta)\backslash\mathrm{tv}(\Gamma, \Gamma'))$$

That is, we can assume that the generalized type variable α should always occur in Δ. For the simplicity of the following proof, we will use $\forall I'$ instead of $\forall I$.

Lemma 8.2

$$tv([\tau/\alpha]) \cap (tv(\Delta)\backslash tv(\Gamma, \Gamma')) = \emptyset \ \wedge \ \Gamma \vdash \Delta \ \textbf{gen} \ \Gamma' \quad \Rightarrow \quad [\tau/\alpha]\Gamma \vdash [\tau/\alpha]\Delta \ \textbf{gen} \ [\tau/\alpha]\Gamma'$$

Proof Induction on the derivation of $\Gamma \vdash \Delta \ \textbf{gen} \ \Gamma'$. The case of (REFL) is trivial. Consider a derivation where the last step is an application of ($\forall I'$):

$$(\forall\text{I'}) \ \frac{\Gamma \vdash \Delta \ \textbf{gen} \ (x : \sigma, \Gamma')}{\Gamma \vdash \Delta \ \textbf{gen} \ (x : \forall\alpha'.\sigma, \Gamma')} \ (\alpha' \in \mathrm{tv}(\Delta)\backslash\mathrm{tv}(\Gamma, \Gamma'))$$

Note that $tv([\tau/\alpha]) = tv(\tau) \cup \{\alpha\}$. We have,

$$\begin{array}{lll}
 & (tv(\Delta)\backslash tv(\Gamma, x : \sigma, \Gamma')) \subseteq (tv(\Delta)\backslash tv(\Gamma, x : \forall\alpha'.\sigma, \Gamma')) & \\
 & \wedge \ tv(\tau) \cap (tv(\Delta)\backslash tv(\Gamma, x : \forall\alpha'.\sigma, \Gamma')) = \emptyset & \textit{assumption} \\
\Rightarrow & tv(\tau) \cap (tv(\Delta)\backslash tv(\Gamma, x : \sigma, \Gamma')) = \emptyset & \\
\Rightarrow & [\tau/\alpha]\Gamma \vdash [\tau/\alpha]\Delta \ \textbf{gen} \ [\tau/\alpha](x : \sigma, \Gamma') & \textit{IH}
\end{array}$$

$$\begin{array}{rcll} \alpha' \in tv(\Delta)\backslash tv(\Gamma, x:\forall\alpha'.\sigma, \Gamma') & \Rightarrow & \alpha' \notin tv(\tau) & \textit{assumption} \\ & \Rightarrow & \alpha' \notin tv([\tau/\alpha]\Gamma, [\tau/\alpha]\Gamma') & \\ \{\alpha\} \cap (tv(\Delta)\backslash tv(\Gamma, x:\sigma, \Gamma')) = \emptyset & \Rightarrow & \alpha \notin tv(\Delta) & \\ \alpha' \in tv(\Delta) \ \wedge \ \alpha \notin tv(\Delta) & \Rightarrow & \alpha \neq \alpha' & \\ \alpha' \in tv(\Delta) \ \wedge \ \alpha \neq \alpha' & \Rightarrow & \alpha' \in tv([\tau/\alpha]\Delta) & \end{array}$$

$$\begin{array}{lll} & [\tau/\alpha]\Gamma \ \vdash \ [\tau/\alpha]\Delta \ \ \mathbf{gen} \ \ [\tau/\alpha](x:\sigma, \Gamma') \ \wedge \ \alpha' \in \mathrm{tv}([\tau/\alpha]\Delta)\backslash \mathrm{tv}([\tau/\alpha](\Gamma, \Gamma')) & \\ \Rightarrow & [\tau/\alpha]\Gamma \ \vdash \ [\tau/\alpha]\Delta \ \ \mathbf{gen} \ \ [\tau/\alpha](x:\forall\alpha'.\sigma, \Gamma') & \forall I' \end{array}$$

□

Lemma 8.3 $\Gamma \ \vdash \ M : \diamond \ \Rightarrow \ [\tau/\alpha]\Gamma \ \vdash \ M : \diamond$

Proof Induction on the size of M. Analyse according to the form of M.

Case $M \equiv u\overline{v}$. We infer as follows.

$$\begin{array}{rcll} \Gamma \ \vdash \ u\overline{v} : \diamond & \Rightarrow & \Gamma \ \vdash \ u : \overline{\tau}_a \to \diamond \ \wedge \ \Gamma \ \vdash \ \overline{v} : \overline{\tau}_a & \\ & \Rightarrow & [\tau/\alpha]\Gamma \ \vdash \ u : [\tau/\alpha]\overline{\tau}_a \to \diamond \ \wedge \ [\tau/\alpha]\Gamma \ \vdash \ v_i : [\tau/\alpha]\tau_a^i & \textit{Lemma } 8.1 \\ & \Rightarrow & [\tau/\alpha]\Gamma \ \vdash \ (u\overline{v}) : \diamond & \end{array}$$

Case $M \equiv M_1 \& M_2$. By induction.

Case $M \equiv \mathbf{def}\,(D_1, .., D_n)$ $\mathbf{in}$ N. Then

$$\begin{array}{ll} & \Gamma \ \vdash \ \mathbf{def}\,(D_1, .., D_n) \ \mathbf{in} \ N : \diamond \\ \Rightarrow & \Gamma \oplus \Delta \ \vdash \ D_i : \Delta_i \ \wedge \ \Gamma \oplus \Gamma' \ \vdash \ \Delta_i \ \ \mathbf{gen} \ \ \Gamma'_i \ \wedge \ \Gamma \oplus \Gamma' \ \vdash \ N : \diamond \\ \Rightarrow & (tv(\Delta)\backslash tv(\Gamma, \Gamma') \cap (tv(\tau) \cup \{\alpha\}) = \emptyset \end{array}$$

The last claim is valid because we can always choose type variables in $tv(\Delta)\backslash tv(\Gamma, \Gamma')$ to be different from those in $(tv(\tau) \cup \{\alpha\})$. It follows that

$$\Gamma \oplus \Gamma' \ \vdash \ \Delta_i \ \ \mathbf{gen} \ \ \Gamma'_i \quad \Rightarrow \quad [\tau/\alpha](\Gamma \oplus \Gamma') \ \vdash \ [\tau/\alpha]\Delta_i \ \ \mathbf{gen} \ \ [\tau/\alpha]\Gamma'_i \qquad \textit{Lemma } 8.2$$

Assume $D_i \equiv x_1(\overline{y}_1) \ \& \ \ldots \ \& \ x_n(\overline{y}_n) = N'$ and $\Delta_i \equiv (x_1 : (\overline{\tau}_1) \to \diamond, \ldots, x_n : (\overline{\tau}_n) \to \diamond)$, then

$$\begin{array}{lll} & \Gamma \oplus \Delta \ \vdash \ D_i : \Delta_i & \\ \Rightarrow & \Gamma \oplus \Delta \oplus (\overline{y}_1 : \overline{\tau}_1, \ldots, \overline{y}_n : \overline{\tau}_n) \ \vdash \ N' : \diamond & \textit{EQN} \\ \Rightarrow & [\tau/\alpha](\Gamma \oplus \Delta \oplus (\overline{y}_1 : \overline{\tau}_1, \ldots, \overline{y}_n : \overline{\tau}_n)) \ \vdash \ N' : \diamond & \textit{IH} \\ \Rightarrow & [\tau/\alpha](\Gamma \oplus \Delta) \ \vdash \ x_1(\overline{y}_1) \ \& \ \ldots \ \& \ x_n(\overline{y}_n) = N' : [\tau/\alpha](x_1 : (\overline{\tau}_1) \to \diamond, \ldots, x_n : (\overline{\tau}_n) \to \diamond) & \textit{EQN} \\ \Rightarrow & [\tau/\alpha](\Gamma \oplus \Delta) \ \vdash \ D_i : [\tau/\alpha]\Delta_i & \end{array}$$

Furthermore,

$$\Gamma \oplus \Gamma' \ \vdash \ N : \diamond \quad \Rightarrow \quad [\tau/\alpha](\Gamma \oplus \Gamma') \ \vdash \ N : \diamond \quad \textit{IH}$$

Apply rule DEF, we get

$$\begin{array}{ll} & [\tau/\alpha](\Gamma \oplus \Delta) \ \vdash \ D_i : [\tau/\alpha]\Delta_i \ \wedge \ [\tau/\alpha](\Gamma \oplus \Gamma') \ \vdash \ [\tau/\alpha]\Delta_i \ \ \mathbf{gen} \ \ [\tau/\alpha]\Gamma'_i \ \wedge \ [\tau/\alpha](\Gamma \oplus \Gamma') \ \vdash \ N : \diamond \\ \Rightarrow & [\tau/\alpha]\Gamma \ \vdash \ \mathbf{def}\,(D_1, .., D_n) \ \mathbf{in} \ N : \diamond \end{array}$$

□

The last three results can be summarised in the following proposition.

Proposition 8.4 (Stability)

1. $\Gamma \vdash x : \sigma$ implies $\phi\Gamma \vdash x : \phi\sigma$
2. $\Gamma \vdash \Delta$ **gen** Γ' and $\mathrm{tv}(\phi) \cap \mathrm{tv}(\Delta) \subseteq \mathrm{tv}(\Gamma, \Gamma')$ imply $\phi\Gamma \vdash \phi\Delta$ **gen** $\phi\Gamma'$
3. $\Gamma \vdash M : \diamond$ implies $\phi\Gamma \vdash M : \diamond$
4. $\Gamma \vdash D : \Delta$ implies $\phi\Gamma \vdash D : \phi\Delta$
5. $\Gamma \vdash D \;:_{\mathrm{gen}}\; \Gamma'$ implies $\phi\Gamma \vdash D \;:_{\mathrm{gen}}\; \phi\Gamma'$

Proof 1. By Lemma 8.1.

2. By Lemma 8.2. Observe that $\mathrm{tv}(\phi) \cap (tv(\Delta)\backslash tv(\Gamma, \Gamma')) = \emptyset \;\Leftrightarrow\; \mathrm{tv}(\phi) \cap \mathrm{tv}(\Delta) \subseteq \mathrm{tv}(\Gamma, \Gamma')$.
3. By Lemma 8.3.
4. Apply ϕ to the antecedent of the rule EQN, then use 3.
5. Apply ϕ to the antecedent of the rule DEF', then use 2 and 4.

□

9 Proof for Type Soundness

This section contains the technical proofs for the type soundness.

Lemma 9.1 (Typing invariant under renaming)

$$\Gamma \vdash z : \tau \;\wedge\; \Gamma, y : \tau \vdash M : \diamond \;\Rightarrow\; \Gamma \vdash [z/y]M : \diamond$$

Lemma 9.2 (AC conversion)

$$\begin{array}{lcl}
\Gamma \vdash M_1 \& M_2 : \diamond & \Leftrightarrow & \Gamma \vdash M_2 \& M_1 : \diamond \\
\Gamma \vdash M_1 \& (M_2 \& M_3) : \diamond & \Leftrightarrow & \Gamma \vdash (M_1 \& M_2) \& M_3 : \diamond \\
\Gamma \vdash (D_1, D_2) \;:_{\mathrm{gen}}\; \Gamma_D & \Leftrightarrow & \Gamma \vdash (D_2, D_1) \;:_{\mathrm{gen}}\; \Gamma_D \\
\Gamma \vdash D_1, (D_2, D_3) \;:_{\mathrm{gen}}\; \Gamma_D & \Leftrightarrow & \Gamma \vdash (D_1, D_2), D_4 \;:_{\mathrm{gen}}\; \Gamma_D \\
\Gamma \vdash D, \epsilon \;:_{\mathrm{gen}}\; \Gamma_D & \Leftrightarrow & \Gamma \vdash D \;:_{\mathrm{gen}}\; \Gamma_D
\end{array}$$

Proof The proofs for these results are straightforward. The first two are by proven by observing the rule PAR, the rest by observing the rule DEF. □

Lemma 9.3 (Scope Extrusion) Assume $dv(D) \cap fv(N) = \emptyset$, then

$$\Gamma \vdash (\mathbf{def}\, D \,\mathbf{in}\, M) \;\&\; N : \diamond \;\Leftrightarrow\; \Gamma \vdash \mathbf{def}\, D \,\mathbf{in}\, (M \;\&\; N) : \diamond$$

Proof Let D be $D_1, .., D_n$ for $n \geq 1$ and Γ' be $\Gamma'_1 \cup ... \cup \Gamma'_n$.

($\Rightarrow$). Assume $\Gamma \vdash (\mathbf{def}\, D \,\mathbf{in}\, M) \;\&\; N : \diamond$,

$$\begin{array}{lll}
 & \Gamma \vdash (\mathbf{def}\, D \,\mathbf{in}\, M) \;\&\; N : \diamond & \\
\Rightarrow & \Gamma \vdash \mathbf{def}\, D \,\mathbf{in}\, M : \diamond \;\wedge\; \Gamma \vdash N : \diamond & PAR \\
\Rightarrow & \Gamma \oplus \Delta \vdash D_i : \Delta_i \;\wedge\; \Gamma \oplus \Gamma' \vdash \Delta_i \;\mathbf{gen}\; \Gamma'_i \;\wedge\; \Gamma \oplus \Gamma' \vdash M : \diamond & DEF \\
 & \wedge\, \Gamma \oplus \Gamma' \vdash N : \diamond & Lemma\ 10.9 \\
\Rightarrow & \Gamma \oplus \Gamma' \vdash M \;\&\; N : \diamond & PAR \\
\Rightarrow & \Gamma \vdash \mathbf{def}\, D \,\mathbf{in}\, (M \;\&\; N) : \diamond & DEF
\end{array}$$

($\Leftarrow$). Assume $\Gamma \vdash \mathbf{def}\, D \,\mathbf{in}\, (M \;\&\; N) : \diamond$ and $dv(D) \cap fv(N) = \emptyset$.

$$\begin{array}{lll}
 & \Gamma \vdash \mathbf{def}\, D \,\mathbf{in}\, (M \;\&\; N) : \diamond & \\
\Rightarrow & \Gamma \oplus \Delta \vdash D_i : \Delta_i \;\wedge\; \Gamma \oplus \Gamma' \vdash \Delta_i \;\mathbf{gen}\; \Gamma'_i \;\wedge\; \Gamma \oplus \Gamma' \vdash M \;\&\; N : \diamond & DEF \\
\Rightarrow & \Gamma \oplus \Gamma' \vdash M : \diamond \;\wedge\; \Gamma \oplus \Gamma' \vdash N : \diamond & PAR \\
\Rightarrow & \Gamma \vdash \mathbf{def}\, D \,\mathbf{in}\, M : \diamond & DEF
\end{array}$$

$$
\begin{array}{lll}
 & dv(D) = dom(\Gamma') \ \wedge \ dv(D) \cap fv(N) = \emptyset & \textit{Lemma } 9.6 \\
\Rightarrow & fv(N) \cap dom(\Gamma') = \emptyset & \\
\Rightarrow & \Gamma \ \vdash \ N : \diamond & \textbf{??} \\
\Rightarrow & \Gamma \ \vdash \ (\mathbf{def}\, D \,\mathbf{in}\, M) \,\&\, N : \diamond & PAR
\end{array}
$$

□

Proof (of Theorem 5.1). The result follows from Lemma 9.2, Lemma 9.3 and Lemma 5.3.

□

Proof (of Lemma 5.5). Induction on the structure of R. In the following, Δ and Γ' denote $\Delta_1 \cup ... \cup \Delta_n$ and $\Gamma'_1 \cup ... \cup \Gamma'_n$ respectively.

$(\Rightarrow)$
Case $R = [\,]$. Straightforward.

Case $R = \mathbf{def}\, D \,\mathbf{in}\, R_1$. Then,

$$
\begin{array}{lll}
 & \Gamma \ \vdash \ \mathbf{def}\, D \,\mathbf{in}\, R_1[M] {:} \diamond & \\
\Rightarrow & \Gamma \oplus \Delta \ \vdash \ D_i : \Delta_i \ \wedge \ \Gamma \oplus \Gamma' \ \vdash \ \Delta_i \ \mathbf{gen} \ \Gamma'_i \ \wedge \ \Gamma \oplus \Gamma' \ \vdash \ R_1[M] : \diamond & DEF \\
\Rightarrow & \Gamma \ \vdash \ D : \Gamma' \ \wedge \ \exists \Gamma_{R_1}.\, (\Gamma \oplus \Gamma' \ \vdash \ R_1 \ :_{\mathrm{gen}} \ \Gamma_{R_1} \ \wedge \ \Gamma \oplus \Gamma', \Gamma_{R_1} \ \vdash \ M : \diamond) & DEF' \ \& \ \textit{Ind.} \\
\Rightarrow & \Gamma \ \vdash \ \mathbf{def}\, D \,\mathbf{in}\, R_1 \ :_{\mathrm{gen}} \ \Gamma', \Gamma_{R_1} & R_{[\,]}
\end{array}
$$

Case $R = R_1 \& M_1$. Then,

$$
\begin{array}{llll}
\Gamma \ \vdash \ R_1[M] \& M_1 : \diamond & \Rightarrow & \Gamma \ \vdash \ R_1[M] : \diamond & PAR \\
 & \Rightarrow & \Gamma \ \vdash \ R_1 \ :_{\mathrm{gen}} \ \Gamma_{R_1} \ \wedge \ \Gamma, \Gamma_{R_1} \ \vdash \ M : \diamond & \textit{Ind.} \\
 & \Rightarrow & \Gamma \ \vdash \ R_1 \& M_1 : \Gamma_{R_1} & R_{\&}
\end{array}
$$

$(\Leftarrow)$
Case $R = [\,]$. Straightforward.

Case $R = \mathbf{def}\, D \,\mathbf{in}\, R_1$. Then Γ_R can be written as Γ', Γ'' where $\Gamma' \equiv \Gamma'_1 \cup ... \cup \Gamma'_n$, we have

$$
\begin{array}{rlll}
\Gamma \ \vdash \ \mathbf{def}\, D \,\mathbf{in}\, R_1 \ :_{\mathrm{gen}} \ \Gamma', \Gamma'' & \Rightarrow & \Gamma \ \vdash \ D \ :_{\mathrm{gen}} \ \Gamma' \ \wedge \ \Gamma, \Gamma' \ \vdash \ R_1 \ :_{\mathrm{gen}} \ \Gamma'' & R_{[\,]} \\
\Gamma, \Gamma', \Gamma'' \ \vdash \ M : \diamond \ \wedge \ \Gamma, \Gamma' \ \vdash \ R_1 \ :_{\mathrm{gen}} \ \Gamma'' & \Rightarrow & \Gamma, \Gamma' \ \vdash \ R_1[M] & \textit{Ind.} \\
\Gamma \ \vdash \ D \ :_{\mathrm{gen}} \ \Gamma' & \Rightarrow & \Gamma \oplus \overline{\Delta} \ \vdash \ D_i : \Gamma_i \ \wedge \ \Gamma \oplus \Gamma' \ \vdash \ \Delta_i \ \mathbf{gen} \ \Gamma'_i & DEF' \\
 & \Rightarrow & \Gamma \ \vdash \ \mathbf{def}\, D \,\mathbf{in}\, R_1[M] : \diamond & DEF
\end{array}
$$

Case $R = R_1 \& M_1$. Then,

$$
\begin{array}{rlll}
\Gamma \ \vdash \ R_1 \& M_1 \ :_{\mathrm{gen}} \ \Gamma_R & \Rightarrow & \Gamma \ \vdash \ R_1 \ :_{\mathrm{gen}} \ \Gamma_R & R_{\&} \\
\Gamma \ \vdash \ R_1 \ :_{\mathrm{gen}} \ \Gamma_R \ \wedge \ \Gamma, \Gamma_R \ \vdash \ M : \diamond & \Rightarrow & \Gamma \ \vdash \ R_1[M] \ :_{\mathrm{gen}} \ \Gamma_R & \textit{Ind.} \\
\Gamma \ \vdash \ R_1[M] \ :_{\mathrm{gen}} \ \Gamma_R & \Rightarrow & \Gamma \ \vdash \ R_1[M] \& M_1 \ :_{\mathrm{gen}} \ \Gamma_R & PAR
\end{array}
$$

□

A few additional structural lemmas are needed in the proofs of subject reduction.

The following lemma shows that typing will be preserved under the polymorphic generalization on any part of the environment.

Lemma 9.4 (Generalization of Environment) Assume that $\Gamma \ \vdash \ \Delta \ \mathbf{gen} \ \Gamma'$, then

$$
\begin{array}{lll}
\Gamma \oplus \Delta \oplus \Gamma'' \ \vdash \ x : \sigma & \Rightarrow & \Gamma \oplus \Gamma' \oplus \Gamma'' \ \vdash \ x : \sigma \\
\Gamma \oplus \Delta \oplus \Gamma'' \ \vdash \ M : \diamond & \Rightarrow & \Gamma \oplus \Gamma' \oplus \Gamma'' \ \vdash \ M : \diamond \\
\Gamma \oplus \Delta \oplus \Gamma'' \ \vdash \ D : \Delta & \Rightarrow & \Gamma \oplus \Gamma' \oplus \Gamma'' \ \vdash \ D : \Delta
\end{array}
$$

Lemma 9.5 (Exchange of subenvironment)

$$\begin{array}{lll} \Gamma \oplus \Gamma_1, \Gamma_2 \vdash M : \diamond & \Rightarrow & \Gamma, \Gamma_2, \Gamma_1 \vdash M : \diamond \\ \Gamma \oplus \Gamma_1, \Gamma_2 \vdash D : \Delta & \Rightarrow & \Gamma, \Gamma_2, \Gamma_1 \vdash D : \Delta \\ \Gamma \oplus \Gamma_1, \Gamma_2 \vdash \Delta \ \mathbf{gen} \ \Gamma'' & \Rightarrow & \Gamma, \Gamma_2, \Gamma_1 \vdash \Delta \ \mathbf{gen} \ \Gamma'' \end{array}$$

Lemma 9.6 (DEF)

$$\begin{array}{ll} & \Gamma \vdash \mathbf{def}\,(D_1, .., D_n)\ \mathbf{in}\ M : \diamond \\ \Rightarrow & \Gamma \oplus \Delta \vdash D_i : \Delta_i \ \wedge \ \Gamma \oplus \Gamma' \vdash \Delta_i \ \mathbf{gen} \ \Gamma'_i \ \wedge \ \Gamma \oplus \Gamma' \vdash M : \diamond \\ & \wedge \ dom(\Gamma') = dom(\Delta) = dv(D_1, .., D_n) \end{array}$$

Proof Simultaneous induction on these three judgements. □

There is a subtle issue in the proof of subject reduction, which concerns the invariance of typing under renaming substitution. Consider a term $\mathbf{def}\ xy = M\ \mathbf{in}\ xz$ and a renaming $\theta \equiv [z/y]$. The reduction will replace xz, which is equal to $\theta(xy)$, by θM. We are required to show that, under certain environment Γ, which includes the typing for x, the derivability of $\Gamma, y : \tau \vdash M : \diamond$ implies that $\Gamma \vdash \theta M : \diamond$. In such a case, one might expect that z, y would be of the same type under the current environment Γ. This might not always be true as x could have a polymorphic type, say $x : \forall\overline{\alpha}.(\tau \rightarrow \diamond) \in \Gamma$. Hence, the typability of xz might not imply that $z : \tau$ but $z : [\overline{\tau'}/\overline{\alpha}]\tau$ for some $\overline{\tau'}$.

A natural question that arises is wether typing could be preserved under such a substitution, that is, the substitution of names $\overline{y}$ of types $\overline{\tau}$ by names $\overline{z}$ of types $[\overline{\tau'}/\overline{\alpha}]\overline{\tau}$. Lemma 9.7 shows that under certain conditions, typability can be preserved under such a substitution. This result can be proved by using the fact that the typability is invariant under a type subsitution for the type variables in the environment (Lemma 8.1,Lemma 8.2,Lemma 8.3).

Lemma 9.7

$$\Gamma, \overline{y} : \overline{\tau} \vdash M : \diamond \ \wedge \ |\overline{\alpha}| \cap \ tv(\Gamma) = \emptyset \ \wedge \ \Gamma \vdash \overline{z} : [\overline{\tau'}/\overline{\alpha}]\overline{\tau} \quad \Rightarrow \quad \Gamma \vdash [\overline{z}/\overline{y}]M : \diamond$$

Proof

$$\begin{array}{llll} \Gamma, \overline{y} : \overline{\tau} \vdash M : \diamond \ \wedge \ |\overline{\alpha}| \cap \ tv(\Gamma) = \emptyset & \Rightarrow & \Gamma, \overline{y} : [\overline{\tau'}/\overline{\alpha}]\overline{\tau} \vdash M : \diamond & \textit{Lemma}\ 8.3 \\ & \Rightarrow & \Gamma \vdash [\overline{z}/\overline{y}]M : \diamond & \textit{Lemma}\ 9.1 \end{array}$$

□

Lemma 9.8 [Property of gen] Let $\Delta \equiv (x_1{:}\overline{\tau}_1, .., x_n{:}\overline{\tau}_n), \Gamma' \equiv (x_1{:}\forall\overline{\alpha}_i.\overline{\tau}_1, .., x_n{:}\forall\overline{\alpha}_n.\overline{\tau}_n)$. Then,

$$\Gamma \oplus \Gamma' \vdash \Delta \ \mathbf{gen} \ \Gamma' \ \wedge \ i \neq j \ \Rightarrow \ \overline{\alpha}_i \notin tv(\overline{\tau}_j)$$

Proof Induction on the derivation of $\Gamma \oplus \Gamma' \vdash \Delta \ \mathbf{gen} \ \Gamma'$. □

Now we can prove the key result for the proof of subject reduction.

Lemma 9.9 (Key Step Subject Reduction)

$$\Gamma \vdash \mathbf{def}\, D, L = M\ \mathbf{in}\ R[\theta L] : \diamond \ \Rightarrow \ \Gamma \vdash \mathbf{def}\, D, L = M\ \mathbf{in}\ R[\theta M] : \diamond$$

Proof (Proof of Lemma 9.9). In the following, each pair of Γ, Γ' will have disjoint domains, so the notation $\Gamma \oplus \Gamma'$ will be same as Γ, Γ'. Let L be $x_1(\overline{y}_1)\ \&\ \ldots\ \&\ x_p(\overline{y}_p)$. Assume $\theta = \{\overline{z}_1/\overline{y}_1, \ldots, \overline{z}_p/\overline{y}_p\}$. Let D be $D_1, .., D_{n-1}$ for $n \geq 1$, D_n be $L = M$, Γ' be $\Gamma'_1 \cup ... \cup \Gamma'_n$ and Δ be $\Delta_1 \cup ... \cup \Delta_n$. Then,

$$\begin{array}{lll} & \Gamma \vdash \mathbf{def}\, D, L = M\ \mathbf{in}\ R[\theta L] : \diamond & \\ \Rightarrow & \Gamma \oplus \Delta \vdash D_i : \Delta_i \ \wedge \ \Gamma \oplus \Gamma' \vdash \Delta_i \ \mathbf{gen} \ \Gamma'_i \ \wedge \ \Gamma \oplus \Gamma' \vdash R[\theta L] : \diamond & \textit{DEF} \\ \Rightarrow & \Gamma \vdash D \ :_{\mathrm{gen}} \ \Gamma' & \end{array}$$

$$\begin{array}{lll} & \Gamma \oplus \Delta \vdash L = M : \Delta_n & \\ \Rightarrow & \Delta_n \equiv (x_1 : \overline{\tau}_1 \rightarrow \diamond, .., x_p : \overline{\tau}_p \rightarrow \diamond) \ \wedge \ \Gamma \oplus \Delta \oplus (\overline{y}_1 : \overline{\tau}_1, .., \overline{y}_p : \overline{\tau}_p) \vdash M : \diamond & \textit{EQN} \\ \Rightarrow & \Gamma \oplus \Gamma' \vdash x_k : \overline{\tau}_k \rightarrow \diamond \ \wedge \ \Gamma \oplus \Gamma' \oplus (\overline{y}_1 : \overline{\tau}_1, .., \overline{y}_p : \overline{\tau}_p) \vdash M : \diamond & \textit{Lemma}\ 9.4 \end{array}$$

$$
\begin{array}{llll}
\Gamma \oplus \Gamma' \vdash R[\theta L] : \diamond & \Rightarrow & \Gamma \oplus \Gamma', \Gamma_R \vdash \theta L : \diamond \wedge \Gamma \oplus \Gamma' \vdash R :_{\text{gen}} \Gamma_R & \textit{Lemma } 5.5 \\
& \Rightarrow & \Gamma \oplus \Gamma', \Gamma_R \vdash x_1(\overline{z}_1) \ \& \ \ldots \ \& \ x_p(\overline{z}_p) : \diamond & \\
& \Rightarrow & \Gamma \oplus \Gamma', \Gamma_R \vdash x_k(\overline{z}_k) : \diamond & \textit{PAR} \\
x_k : \overline{\tau}_k \rightarrow \diamond \in \Delta_n & \Rightarrow & x_k : \forall \overline{\alpha}_k.(\overline{\tau}_k \rightarrow \diamond) \in \Gamma'_n \subseteq \Gamma' & \\
& \Rightarrow & \Gamma \oplus \Gamma', \Gamma_R \vdash \overline{z}_k : [\overline{\tau'}_k / \overline{\alpha}_k]\overline{\tau}_k &
\end{array}
$$

$$
\begin{array}{ll}
& \Gamma \vdash D :_{\text{gen}} \Gamma' \wedge \Gamma \oplus \Gamma' \vdash R :_{\text{gen}} \Gamma_R \wedge x_k : \forall \overline{\alpha}_k.(\overline{\tau}_k \rightarrow \diamond) \in \Gamma' \\
\Rightarrow & |\overline{\alpha}_k| \cap tv(\Gamma \oplus \Gamma', \Gamma_R) = \emptyset
\end{array}
$$

By Lemma 9.8, $\overline{\alpha}_k \cap \overline{\alpha}_h = \emptyset$. Let $[\overline{\tau'}/\overline{\alpha}]$ denotes $[\overline{\tau'}_1/\overline{\alpha}_1, .., \overline{\tau'}_n/\overline{\alpha}_n]$, we have

$$
\begin{array}{llll}
\Gamma \oplus \Gamma' \vdash \Delta_n \ \mathbf{gen} \ \Gamma'_n \wedge k \neq h & \Rightarrow & \overline{\alpha}_h \notin tv(\overline{\tau}_k) & \textit{Lemma } 9.8 \\
& \Rightarrow & [\overline{\tau'}_k/\overline{\alpha}_k]\overline{\tau}_k = [\overline{\tau'}/\overline{\alpha}]\overline{\tau}_k & \\
\Gamma \oplus \Gamma', \Gamma_R \vdash \overline{z}_k : [\overline{\tau'}_k/\overline{\alpha}_k]\overline{\tau}_k & \Rightarrow & \Gamma \oplus \Gamma', \Gamma_R \vdash \overline{z}_k : [\overline{\tau'}/\overline{\alpha}]\overline{\tau}_k &
\end{array}
$$

Therefore,

$$
\begin{array}{lll}
& |\overline{\alpha}| \cap tv(\Gamma \oplus \Gamma', \Gamma_R) = \emptyset \wedge \Gamma \oplus \Gamma', \Gamma_R \vdash \overline{z}_k : [\overline{\tau'}/\overline{\alpha}]\overline{\tau}_k & \\
& \wedge \ \Gamma \oplus \Gamma', (\overline{y}_1 : \overline{\tau}_1 .. \overline{y}_p : \overline{\tau}_p) \vdash M : \diamond & \\
\Rightarrow & \Gamma \oplus \Gamma', \Gamma_R \vdash \theta M : \diamond & \textit{Lemma } 9.7 \\
\Rightarrow & \Gamma \oplus \Gamma' \vdash R[\theta M] : \diamond & \textit{Lemma } 5.5 \\
\Rightarrow & \Gamma \vdash \mathbf{def}\, D, L = M \ \mathbf{in}\ R[\theta M] : \diamond &
\end{array}
$$

□

Now the subject reduction can be proved.

Proof (of Theorem 5.2). The result follows from Lemma 9.9 and Lemma 5.3. □

10 Type Inference Proofs

In some places we will write W_x, W_M, W_D, and mgu with an additional result parameter yielding all the fresh variables created in this call. This simplifies the completeness proof. $C(U)$ denotes the complement of U with respect to the set of all variables.

Lemma 10.1 (Most General Unifier)

1. If $\theta' = mgu(\theta E)$ and θ idempotent then θ' idempotent and $\theta'\theta$ idempotent and $\theta'\theta E$ and $\mathrm{dom}(\theta') \subseteq \mathrm{tv}(\theta E)$.
2. If ϕE and $\theta \leq^{\psi}_{C(\mathcal{U})} \phi$ and $\mathrm{tv}(E) \cap \mathcal{U} = \emptyset$, then $(\mathcal{V}, \theta') = mgu(\theta E)$ exists with $\theta'\theta \leq^{\psi'}_{C(\mathcal{V} \cup \mathcal{U})} \phi$.

Definition. We define $\mathrm{dom}(\phi) = \{u | \phi u \neq u\}$, $\mathrm{codom}(\phi) = \bigcup_{u \in \mathrm{dom}(\phi)} \mathrm{tv}(\phi u)$ and $\mathrm{tv}(\phi) = \mathrm{dom}(\phi) \cup \mathrm{codom}(\phi)$.

Lemma 10.2 (Comparison of Substitutions) If $\phi \leq^{\chi}_{U} \psi$ then

1. $\chi\phi = \psi$.
2. $\chi\psi = \chi = \psi\chi$.

Lemma 10.3 (transitivity) If $\theta_1 \leq^{\psi_1}_{U} \psi_0$ and $\theta_2 \leq^{\psi_2}_{V} \psi_1$ then $\theta_2 \leq^{\psi_2}_{U \cap V} \psi_0$.

Lemma 10.4 (reflexivity) If $\theta_1 \leq^{\psi_1}_{U} \psi_0$ then $\theta_1 \leq^{\psi_1}_{C(\emptyset)} \psi_1$.

Lemma 10.5 (extension) If $\psi\theta = \psi$ and $\mathrm{dom}(\psi') \cap \mathrm{tv}(\psi) = \emptyset$ and $\psi|_{\mathrm{codom}(\psi')} = id$ and ψ, ψ' and θ are idempotent then $\theta \leq^{\psi'\psi}_{C(\mathrm{dom}(\psi'))} \psi$.

Lemma 10.6 (Generic Instance) The generic instance relation is reflexive, transitive and stable under substitution.

Proof

1. $\sigma \subseteq \sigma'$ implies $\phi\sigma \subseteq \phi\sigma'$
   ASSUME: $\forall\overline{\alpha}.\tau \subseteq \forall\overline{\beta}.\tau'$
   PROVE: $\phi\forall\overline{\alpha}.\tau \subseteq \phi\forall\overline{\beta}.\tau'$
   1.1. Choose $\overline{\tau}''$ such that $[\overline{\tau}''/\overline{\alpha}]\tau = \tau'$
      PROOF: definition generic instance
   1.2. $[\phi\overline{\tau}''/\overline{\alpha}]\phi\tau = \phi\tau'$
      PROOF: $\phi\alpha = \alpha$
   1.3. $\forall\overline{\alpha}.\phi\tau \subseteq \forall\overline{\beta}.\phi\tau'$
      PROOF: definition generic instance
   1.4. Q.E.D.
      PROOF: $\phi\overline{\alpha} = \overline{\alpha} = \phi\overline{\beta} = \overline{\beta}$
2. $\sigma \subseteq \sigma$
   PROVE: $\forall\overline{\alpha}.[\overline{\alpha}/\overline{\beta}]\tau \subseteq \forall\overline{\beta}.\tau$
   2.1. $[\overline{\beta}/\overline{\alpha}][\overline{\alpha}/\overline{\beta}]\tau = \tau$
3. $\sigma_1 \subseteq \sigma_2$ and $\sigma_2 \subseteq \sigma_3$ imply $\sigma_1 \subseteq \sigma_3$
   ASSUME: 1. $\sigma_1 \subseteq \sigma_2$
   2. $\sigma_2 \subseteq \sigma_3$
   PROVE: $\sigma_1 \subseteq \sigma_3$
   3.1. Choose $\forall\overline{\alpha}_i.\tau_i = \sigma_i$
   3.2. $\phi_1\tau_1 = \tau_2$
      PROOF: assumption 1
   3.3. $\phi_2\tau_2 = \tau_3$
      PROOF: assumption 2
   3.4. $\sigma_1 \subseteq \sigma_3$
      PROOF: $\phi_2\phi_1\tau_1 = \tau_3$

Lemma 10.7 (Generalization)

$$\begin{array}{lcl}
\phi gen(\Gamma, \bigcup_i mtv(\Delta_i, \mathrm{dn}(L_i)), \Delta_j) & \subseteq & gen(\phi\Gamma, \bigcup_i mtv(\phi\Delta_i, \mathrm{dn}(L_i)), \phi\Delta_j) \\
\forall i.\ \Gamma'_i = gen(\Gamma, \bigcup_j mtv(\Delta_j, \mathrm{dn}(L_j)), \Delta_i) & \Rightarrow & \forall i.\ \Gamma \oplus (\Gamma'_1 \cup \ldots \cup \Gamma'_n) \ \vdash\ \Delta_i \ \mathbf{gen}\ \Gamma'_i \\
\forall i.\ \Gamma \oplus (\Gamma'_1 \cup \ldots \cup \Gamma'_n) \ \vdash\ \Delta_i \ \mathbf{gen}\ \Gamma'_i & \Rightarrow & \forall i.\ gen(\Gamma, \bigcup_j mtv(\Delta_j, \mathrm{dn}(L_j)), \Delta_i) \subseteq \Gamma'_i
\end{array}$$

Proof

1. Statement 1
   1.1. $gen(\Gamma, \bigcup_i mtv(\Delta_i, \mathrm{dn}(L_i)), \Delta_j) \subseteq \Delta_j$
   1.2. $\phi gen(\Gamma, \bigcup_i mtv(\Delta_i, \mathrm{dn}(L_i)), \Delta_j) \subseteq \phi\Delta_j$
      PROOF: substitution with ϕ, Lemma 10.6;1
   1.3. $\mathrm{tv}(\phi mtv(\Delta_i, \mathrm{dn}(L_i))) \subseteq mtv(\phi\Delta_i, \mathrm{dn}(L_i))$
      PROOF: definition of mtv, $\mathrm{tv}(\phi(\mathrm{tv}(\Delta_i(x_k)) \cap \mathrm{tv}(\Delta_i(x_l)))) \subseteq \mathrm{tv}(\phi\Delta_i(x_k)) \cap \mathrm{tv}(\phi\Delta_i(x_l))$
   1.4. $\mathrm{tv}(\phi gen(\Gamma, \bigcup_i mtv(\Delta_i, \mathrm{dn}(L_i)), \Delta_j)) \subseteq \mathrm{tv}(\phi\Gamma) \cup \bigcup_i mtv(\phi\Delta_i, \mathrm{dn}(L_i))$
      PROOF: $\mathrm{tv}(\phi(\mathrm{tv}(\Delta_j) \cap (\mathrm{tv}(\Gamma) \cup \bigcup_i mtv(\Delta_i, \mathrm{dn}(L_i))))) \subseteq \mathrm{tv}(\phi\Gamma) \cup \bigcup_i mtv(\phi\Delta_i, \mathrm{dn}(L_i))$
   1.5. Q.E.D.
2. Statement 2
   ASSUME: $\forall i.\ \Gamma'_i = gen(\Gamma, \bigcup_j mtv(\Delta_j, \mathrm{dn}(L_j)), \Delta_i)$
   PROVE: $\forall j.\ \Gamma \oplus \bigcup_k \Gamma'_k \ \vdash\ \Delta_i \ \mathbf{gen}\ [x_k : \Gamma'_i(x_k)]_{k=1\ldots j} \oplus [x_k : \Delta_i(x_k)]_{k=j+1\ldots l}$
   2.1. Case $j = 0$
      PROOF: (REFL)
   2.2. Case j assuming $j-1$
      ASSUME: $\Gamma \oplus \bigcup_k \Gamma'_k \ \vdash\ \Delta_i \ \mathbf{gen}\ [x_k : \Gamma'_i(x_k)]_{k=1\ldots j-1} \oplus [x_j : \Delta_i(x_j)] \oplus [x_k : \Delta_i(x_k)]_{k=j+1\ldots l}$

PROVE: $\Gamma \oplus \bigcup_k \Gamma'_k \vdash \Delta_i$ **gen** $[x_k : \Gamma'_i(x_k)]_{k=1\ldots j-1} \oplus [x_j : \Gamma'_i(x_j)] \oplus [x_k : \Delta_i(x_k)]_{k=j+1\ldots l}$
PROOF SKETCH: If $\overline{\alpha}$ are the variables which are generalized, we show, that the $\overline{\alpha}$ do not occur in $\mathrm{tv}(\Gamma)$, $\mathrm{tv}(\Gamma'_k)$ or $\mathrm{tv}(\Delta_i(x_k))$ for $k \neq j$.
2.2.1. Choose $mono := \bigcup_m mtv(\Delta_m, \mathrm{dn}(L_m))$
2.2.2. $\overline{\alpha} = \mathrm{tv}(\Delta_i(x_j)) - (\mathrm{tv}(\Gamma) \cup mono)$
PROOF: This is the way, *gen* determines $\overline{\alpha}$
2.2.3. $\overline{\alpha} \cap \mathrm{tv}(\Gamma) = \emptyset$
PROOF: $\mathrm{tv}(\Gamma) \subseteq (\mathrm{tv}(\Gamma) \cup mono)$
2.2.4. $\overline{\alpha} \cap \mathrm{tv}(\Gamma'_k) = \emptyset$
PROOF: $\mathrm{tv}(\Gamma'_k) = \mathrm{tv}(gen(\Gamma, mono, \Delta_k)) \subseteq \mathrm{tv}(\Gamma) \cup mono$
2.2.5. $k \neq j$ implies $\overline{\alpha} \cap \mathrm{tv}(\Delta_i(x_k)) = \emptyset$
ASSUME: $k \neq j$
PROVE: $\overline{\alpha} \cap \mathrm{tv}(\Delta_i(x_k)) = \emptyset$
PROOF: $\mathrm{tv}(\Delta_i(x_j)) \cap \mathrm{tv}(\Delta_i(x_k)) \subseteq mtv(\Delta_i, \mathrm{dn}(L_i)) \subseteq mono$
2.3. Q.E.D.
3. $\Gamma \vdash \Delta$ **gen** Γ' implies
1. $j \neq k$ implies $\mathrm{tv}(\Delta(x_j)) \cap \mathrm{tv}(\Delta(x_k)) \subseteq \mathrm{tv}(\Gamma'(x_j)) \cap \mathrm{tv}(\Gamma'(x_k))$
2. $\mathrm{tv}(\Gamma) \cap \mathrm{tv}(\Delta) \subseteq \mathrm{tv}(\Gamma')$
PROOF: easy inductions on derivation of $\Gamma \vdash \Delta$ **gen** Γ'
4. Statement 3
ASSUME: $\forall i.\ \Gamma \oplus (\Gamma'_1 \cup \ldots \cup \Gamma'_n) \vdash \Delta_i$ **gen** Γ'_i
PROVE: $gen(\Gamma, \bigcup_j mtv(\Delta_j, \mathrm{dn}(L_j)), \Delta_i) \subseteq \Gamma'_i$
PROOF SKETCH: The idea is that variables, that are generalized somewhere (the V_i), are generalized everywhere (the set V). On the other hand variables, that are in $\bigcup mtv(\_)$ or Γ cannot be generalized anywhere.
4.1. Choose $V_i := \mathrm{tv}(\Delta_i) - \mathrm{tv}(\Gamma'_i)$
4.2. Choose $V := \bigcup_i V_i$
4.3. $V_i = \mathrm{tv}(\Delta_i) \cap V$
4.3.1. $V_i \subseteq \mathrm{tv}(\Delta_i) \cap V$
PROOF: subset of both components
4.3.2. $V_i \supseteq \mathrm{tv}(\Delta_i) \cap V$
ASSUME: $\beta \in \mathrm{tv}(\Delta_i), \beta \in V_k, \beta \notin V_i$
PROVE: False
$\langle 4 \rangle 1.$ $\beta \in \mathrm{tv}(\Gamma'_i)$
PROOF: $\mathrm{tv}(\Gamma'_i) = \mathrm{tv}(\Delta_i) - V_i$
$\langle 4 \rangle 2.$ $\beta \in \mathrm{tv}(\Delta_k)$
PROOF: $\mathrm{tv}(\Delta_k) \supseteq V_k$
$\langle 4 \rangle 3.$ $\beta \in \mathrm{tv}(\Gamma'_k)$
PROOF: 3;2 and the assumption give $\mathrm{tv}(\Gamma'_i) \cap \mathrm{tv}(\Delta_k) \subseteq \mathrm{tv}(\Gamma'_k)$
$\langle 4 \rangle 4.$ $\beta \notin V_k$
PROOF: $V_k = \mathrm{tv}(\Delta_k) - \mathrm{tv}(\Gamma'_k)$
4.4. $V \cap \mathrm{tv}(\Gamma) = \emptyset$
ASSUME: $\beta \in V_i, \beta \in \mathrm{tv}(\Gamma)$
PROVE: False
4.4.1. $\beta \in \mathrm{tv}(\Delta_i) \supseteq V_i$
4.4.2. $\beta \in \mathrm{tv}(\Gamma'_i)$
PROOF: 3;2 gives $\mathrm{tv}(\Gamma) \cap \mathrm{tv}(\Delta_i) \subseteq \mathrm{tv}(\Gamma'_i)$
4.4.3. $\beta \notin \mathrm{tv}(\Gamma'_i)$
PROOF: $\mathrm{tv}(\Gamma'_i) = \mathrm{tv}(\Delta_i) - V_i$
4.5. $V \cap mtv(\Delta_i, \mathrm{dn}(L_i)) = \emptyset$
ASSUME: $\beta \in mtv(\Delta_i, \mathrm{dn}(L_i))$ and $\beta \in V$
PROVE: False
4.5.1. Choose $k \neq l$ with $\beta \in \mathrm{tv}(\Delta_i(x_k)) \cap \mathrm{tv}(\Delta_i(x_l))$
PROOF: definition *mtv*
4.5.2. $\beta \in \mathrm{tv}(\Delta_i) \cap V = V_i$
PROOF: 4.3, 4.5.1 and assumption
4.5.3. $\beta \in \mathrm{tv}(\Gamma'_i(x_k)) \cap \mathrm{tv}(\Gamma'_i(x_l))$
PROOF: 3;1

4.5.4. $\beta \notin V_i$
PROOF: definition V_i (4.1) and $\beta \in \text{tv}(\Gamma'_i)$
4.6. $gen(\Gamma, \bigcup_j mtv(\Delta_j, \text{dn}(L_j)), \Delta_i(x_k)) \subseteq \Gamma'_i(x_k)$
PROOF: $\Gamma'_i(x_k) = \forall V_i.\Delta_i(x_k)$, generic instance, $V_i \cap (\text{tv}(\Gamma) \cup \bigcup_j mtv(\Delta_j, \text{dn}(L_j)) = \emptyset$
4.7. $gen(\Gamma, \bigcup_j mtv(\Delta_j, \text{dn}(L_j)), \Delta_i) \subseteq \Gamma'_i$
PROOF: definition of gen and $\subseteq$
5. Q.E.D.

Lemma 10.8 (Properties)

1. $\sigma \subseteq \sigma'$ implies $\text{tv}(\sigma) \subseteq \text{tv}(\sigma')$
2. $\Gamma \vdash x : \sigma$ implies $\Gamma(x) \subseteq \sigma$
3. $\Gamma(x) \subseteq \tau$ implies $\Gamma \vdash x : \tau$

Proof

1. $\sigma \subseteq \sigma'$ implies $\text{tv}(\sigma) \subseteq \text{tv}(\sigma')$
ASSUME: $\sigma \subseteq \sigma'$
PROVE: $\text{tv}(\sigma) \subseteq \text{tv}(\sigma')$
1.1. $[\overline{\tau}''/\overline{\alpha}]\tau = \tau'$
1.2. $\text{tv}(\sigma) \subseteq \text{tv}(\sigma')$
PROOF: $\text{tv}(\sigma) = \text{tv}(\tau) - \overline{\alpha} - \overline{\beta} \subseteq \text{tv}([\overline{\tau}''/\overline{\alpha}]\tau) - \overline{\beta} = \text{tv}(\tau') - \overline{\beta} = \text{tv}(\sigma')$
2. $\Gamma \vdash x : \sigma$ implies $\Gamma(x) \subseteq \sigma$
ASSUME: $\Gamma \vdash x : \sigma$
PROVE: $\Gamma(x) \subseteq \sigma$
2.1. Case $\sigma = \Gamma(x)$
PROOF: immediate.
2.2. Case $\Gamma \vdash x : \forall\alpha.\sigma'$ and $\sigma = [\tau/\alpha]\sigma'$
PROOF: $\forall\alpha.\sigma' \subseteq [\tau/\alpha]\sigma'$ and transitivity of $\subseteq$
3. $\Gamma(x) \subseteq \tau$ implies $\Gamma \vdash x : \tau$
ASSUME: $\Gamma(x) \subseteq \tau$
PROVE: $\Gamma \vdash x : \tau$
3.1. $\Gamma(x) = \forall\overline{\alpha}.\tau'$ and $[\overline{\tau}''/\overline{\alpha}]\tau' = \tau$
3.2. $\Gamma \vdash x : [\tau''_i/\alpha_i]_{i=j\ldots n}\forall\alpha_{1\ldots j-1}.\tau'$
3.2.1. Case $j = 0$
PROOF: immediate.
3.2.2. Case $j + 1$
PROOF: $(\forall E)$

Lemma 10.9 (Weakening) If $\Gamma' \subseteq \Gamma$ then:

1. $\Gamma \vdash x : \sigma$ and $\sigma \subseteq \tau$ imply $\Gamma' \vdash x : \tau$
2. $\Gamma \vdash \Delta$ **gen** Γ'' implies $\Gamma' \vdash \Delta$ **gen** Γ''
3. $\Gamma \vdash M : \diamond$ implies $\Gamma' \vdash M : \diamond$
4. $\Gamma \vdash D : \Delta$ implies $\Gamma' \vdash D : \Delta$
5. $\Gamma \vdash D :_{\text{gen}} \Gamma''$ implies $\Gamma' \vdash D :_{\text{gen}} \Gamma''$

Proof The proof proceeds by simultaneous structural induction on the derivation of the judgements.

ASSUME: $\Gamma' \subseteq \Gamma$
1. $\Gamma \vdash x : \sigma$ and $\sigma \subseteq \tau$ imply $\Gamma' \vdash x : \tau$
PROOF: $\Gamma'(x) \subseteq \Gamma(x) \subseteq \sigma \subseteq \tau$ by Lemma 10.8;2 and then Lemma 10.8;3
2. $\Gamma \vdash \Delta$ **gen** Γ'' implies $\Gamma' \vdash \Delta$ **gen** Γ''
PROOF: For the induction step we use that $\Gamma' \subseteq \Gamma$ implies $\text{tv}(\Gamma') \subseteq \text{tv}(\Gamma)$ by Lemma 10.8;1
3. $\Gamma \vdash M : \diamond$ implies $\Gamma' \vdash M : \diamond$
PROOF: (APP) and (PAR) are trivial induction steps. For (DEF) the induction step works because $\Gamma' \oplus \Gamma'' \subseteq \Gamma \oplus \Gamma''$ for arbitrary Γ''

4. $\Gamma \vdash D : \Delta$ implies $\Gamma' \vdash D : \Delta$
   PROOF: For the induction step we use again $\Gamma' \oplus \Gamma'' \subseteq \Gamma \oplus \Gamma''$
5. $\Gamma \vdash D :_{\text{gen}} \Gamma''$ implies $\Gamma' \vdash D :_{\text{gen}} \Gamma''$
   PROOF: For the induction step we use again $\Gamma' \oplus \Gamma''' \subseteq \Gamma \oplus \Gamma'''$

Theorem 10.10 (Soundness) If ϕ idempotent then

$$\begin{array}{lll} W_x(\phi, \Gamma, x) = \tau & \Rightarrow & \phi\Gamma \vdash x : \tau \text{ and } \phi\tau = \tau \\ W_M(\phi, \Gamma, M) = \phi' & \Rightarrow & \phi'\Gamma \vdash M : \diamond \\ W_D(\phi, \Gamma, D) = (\phi', \Gamma') & \Rightarrow & \phi'\Gamma \vdash D :_{\text{gen}} \Gamma' \text{ and } \phi'\Gamma' = \Gamma' \end{array}$$

and $\phi' = \phi'\phi$ and ϕ' idempotent, where $\phi' = \phi$ for the first case.

Proof The proof proceeds by simultaneous induction on the three statements on the size of the term to type-check.

1. Statement 1
   ASSUME: 1. $W_x(\phi, \Gamma, x) = \tau$
   2. ϕ idempotent
   PROVE: 1. $\phi\Gamma \vdash x : \tau$
   2. $\phi\tau = \tau$
   1.1. $\phi\tau = \tau$
   PROOF: $\phi\phi inst(\Gamma(x)) = \phi inst(\Gamma(x))$
   1.2. $\phi\Gamma \vdash x : \tau$
   PROOF: We have $\Gamma \vdash x : inst(\Gamma(x))$ by $\Gamma \vdash x : \Gamma(x)$, $\Gamma(x) \subseteq inst(\Gamma(x))$ and weakening and then substitution with ϕ
   1.3. Q.E.D.
2. Statement 2
   ASSUME: 1. $W_M(\phi, \Gamma, M) = \phi'$
   2. ϕ idempotent
   PROVE: 1. $\phi'\Gamma \vdash M : \diamond$
   2. $\phi'\phi = \phi'$
   3. ϕ' idempotent.
   2.1. Case $x\overline{y}$
   2.1.1. Choose $\tau' = W_x(\phi, \Gamma, x)$ with
   1. $\phi\Gamma \vdash x : \tau'$
   2. $\phi\tau' = \tau'$
   PROOF: induction hypothesis
   2.1.2. Choose $\tau_i = W_x(\phi, \Gamma, y_i)$ with
   1. $\phi\Gamma \vdash y_i : \tau_i$
   2. $\phi\tau_i = \tau_i$
   PROOF: induction hypothesis
   2.1.3. $\phi' = mgu(\_)\phi$ idempotent
   PROOF: Lemma 10.1
   2.1.4. $\phi'\phi = \phi'$
   PROOF: $mgu(\_)\phi\phi = mgu(\_)\phi$
   2.1.5. $\phi'\Gamma \vdash x : \phi'\tau'$
   PROOF: 2.1.1;1 and substitution with ϕ'
   2.1.6. $\phi'\Gamma \vdash \overline{y} : \phi'\overline{\tau}$
   PROOF: 2.1.2;1 and substitution with ϕ'
   2.1.7. $\phi'\Gamma \vdash x : \phi'\overline{\tau} \to \diamond$
   PROOF: $\phi'\tau' = mgu(\_)\phi\tau' = mgu(\_)\tau' = mgu(\_)\overline{\tau} \to \diamond = mgu(\_)\phi\overline{\tau} \to \diamond = \phi'\overline{\tau} \to \diamond$ because 2.1.1;2 and 2.1.2;2
   2.1.8. $\phi'\Gamma \vdash x\overline{y} : \diamond$
   PROOF: (APP)
   2.1.9. Q.E.D.
   2.2. Case $M_1 \& M_2$

2.2.1. Choose $\phi'' = W_M(\phi, \Gamma, M_1)$ with
  1. $\phi''\Gamma \vdash M_1 : \diamond$
  2. $\phi''\phi = \phi''$
  3. ϕ'' idempotent.

  Proof: induction hypothesis

2.2.2. $\phi' = W_M(\phi'', \Gamma, M_2)$ with
  1. $\phi'\Gamma \vdash M_2 : \diamond$
  2. $\phi'\phi'' = \phi'$
  3. ϕ' idempotent.

  Proof: 2.2.1;3 and induction hypothesis

2.2.3. $\phi'\phi = \phi'$

  Proof: $\phi'\phi = \phi'\phi''\phi = \phi'\phi'' = \phi'$

2.2.4. $\phi'\Gamma \vdash M_1 : \diamond$

  Proof: 2.2.1;1 and substitution with ϕ'

2.2.5. $\phi'\Gamma \vdash M_1 \& M_2 : \diamond$

  Proof: (PAR)

2.2.6. Q.E.D.

2.3. Case **def** $L_1 = M_1, \ldots, L_n = M_n$ **in** M'

ϕ'' will correspond to ϕ' in the algorithm, because we want to use ϕ' for the result.

2.3.1. Choose $(\phi'', \Gamma') = W_D(\phi, \Gamma, L_1 = M_1, \ldots, L_n = M_n)$ with
  1. $\phi''\Gamma \vdash L_1 = M_1, \ldots, L_n = M_n \ :_{\text{gen}} \ \Gamma'$
  2. $\phi''\Gamma' = \Gamma'$
  3. $\phi''\phi = \phi''$
  4. ϕ'' idempotent

  Proof: induction hypothesis

2.3.2. $\phi' = W_M(\phi'', \Gamma \oplus \Gamma', M')$ with
  1. $\phi'(\Gamma \oplus \Gamma') \vdash M' : \diamond$
  2. $\phi'\phi'' = \phi'$
  3. ϕ' idempotent.

  Proof: induction hypothesis

2.3.3. $\phi'\phi = \phi'$

  Proof: $\phi'\phi = \phi'\phi''\phi = \phi'\phi'' = \phi'$

2.3.4. $\phi'\Gamma \vdash L_1 = M_1, \ldots, L_n = M_n \ :_{\text{gen}} \ \phi'\Gamma'$

  Proof: 2.3.1;1 and substitution with ϕ'

2.3.5. Choose Δ_i, Γ'_i with
  1. $\phi'\Gamma' = \bigcup_i \phi'\Gamma'_i$
  2. $\phi'\Gamma \oplus \phi'\Gamma' \vdash \Delta_i$ **gen** $\phi'\Gamma'_i$
  3. $\phi'\Gamma \oplus \bigcup_i \Delta_i \vdash L_i = M_i : \Delta_i$
  4. $\phi'\Delta_i = \Delta_i$

  Proof: $(DEF')^{-1}$

2.3.6. $\phi'\Gamma \vdash$ **def** $L_1 = M_1, \ldots, L_n = M_n$ **in** $M' : \diamond$

  Proof: (DEF)

2.3.7. Q.E.D.

2.4. Q.E.D.

3. Statement 3

Assume: 1. $W_D(\phi, \Gamma, L_1 = M_1, \ldots, L_n = M_n) = (\phi', \Gamma')$
  2. ϕ idempotent

Prove: 1. $\phi'\Gamma \vdash L_1 = M_1, \ldots, L_n = M_n \ :_{\text{gen}} \ \Gamma'$
  2. $\phi'\phi = \phi'$
  3. ϕ' idempotent
  4. $\phi'\Gamma' = \Gamma'$

3.1. $\phi_0 = mgu(\_)\phi$ idempotent

  Proof: Lemma 10.1

3.2. $\phi_0\phi = \phi_0$

  Proof: $\phi_0\phi = mgu(\_)\phi\phi = mgu(\_)\phi = \phi_0$

3.3. $\phi_i = W_M(\phi_{i-1}, \Gamma \oplus \Delta \oplus lenv(\Delta_i, L_i), M_i)$ with
  1. $\phi_i(\Gamma \oplus \Delta \oplus lenv(\Delta_i, L_i)) \vdash M_i : \diamond$
  2. $\phi_i\phi_{i-1} = \phi_i$

3. ϕ_i idempotent.
PROOF: all ϕ_i idempotent, induction on i, induction hyothesis

3.4. $\phi_i(\Gamma \oplus \Delta) \vdash L_i = M_i : \phi_i\Delta_i$
PROOF: (EQN)

3.5. $\phi' = \phi_n$ idempotent
PROOF: The algorithm returns ϕ_n

3.6. $\phi'\phi_i = \phi'$ for all i
PROOF: induction n to 1: $\phi'\phi_{i-1} = \phi'\phi_i\phi_{i-1} = \phi'\phi_i = \phi'$ and $\phi'\phi_n = \phi'\phi' = \phi'$

3.7. $\phi'\phi = \phi'$
PROOF: $\phi'\phi = \phi'\phi_0\phi = \phi'\phi_0 = \phi'$

3.8. Choose $mono = \bigcup_j mtv(\phi_n\Delta_j, \mathrm{dn}(L_j)) = \bigcup_j mtv(\phi_n\Delta, \mathrm{dn}(L_j))$
PROOF: only the environment on variables from $\mathrm{dn}(L_j)$ matters

3.9. $\Gamma' = gen(\phi_n\Gamma, mono, \phi_n\Delta)$

3.10. Choose $\Gamma'_i = gen(\phi_n\Gamma, mono, \phi_n\Delta_i)$

3.11. $\Gamma' = \bigcup_i \Gamma'_i$

3.12. $\phi'\Gamma' = \Gamma'$, $\phi'\Gamma'_i = \Gamma'_i$
PROOF: $\mathrm{dom}(\phi') \cap \mathrm{tv}(\Gamma'_i) \subseteq \mathrm{dom}(\phi') \cap \mathrm{tv}(\phi'\Delta_i) = \emptyset$

3.13. $\phi'(\Gamma \oplus \Delta) \vdash L_i = M_i : \phi'\Delta_i$
PROOF: 3.4 and substitution with ϕ'

3.14. $\phi'\Delta = \phi'\Delta_1 \cup \ldots \cup \phi'\Delta_n$
PROOF: $\Delta = \phi_0\Delta_1 \cup \ldots \cup \phi_0\Delta_n$ and substitution with ϕ'

3.15. $\phi'\Gamma \oplus \Gamma' \vdash \phi'\Delta_i$ **gen** Γ'_i
PROOF: by Lemma 10.7, 3.8 and 3.10

3.16. $\phi'\Gamma \oplus \bigcup_i \phi'\Delta_i \vdash L_i = M_i : \phi'\Delta_i$
PROOF: by 3.13 and 3.14

3.17. $\phi'\Gamma \vdash L_1 = M_1, \ldots, L_n = M_n \ :_{\mathrm{gen}} \ \Gamma'$
PROOF: (DEF')

3.18. Q.E.D.

Corollary 10.11 (Soundness) $W_M(id, \Gamma, M) = \phi \ \Rightarrow \ \phi\Gamma \vdash M : \diamond$.

Proof Let $\phi = id$ in Theorem 10.10

Theorem 10.12 (Completeness) Assume $\phi \leq^{\chi}_{\mathcal{U}} \psi$. Then

$$\begin{array}{lll}
\psi\Gamma \vdash x : \psi\tau & \Rightarrow & W_x(\phi, \Gamma, x) = (\mathcal{V}, \tau') \text{ and } \chi'\tau' = \chi\tau \\
\psi\Gamma \vdash M : \diamond & \Rightarrow & W_M(\phi, \Gamma, M) = (\mathcal{V}, \phi') \\
\psi\Gamma \vdash D \ :_{\mathrm{gen}} \ \psi\Gamma' & \Rightarrow & W_D(\phi, \Gamma, D) = (\mathcal{V}, \phi', \Gamma'') \text{ and } \chi'\Gamma'' \subseteq \chi\Gamma'
\end{array}$$

for some fresh $\mathcal{V}$ and $\phi' \leq^{\chi'}_{C(\mathcal{V})} \chi$, where $\phi' = \phi$ for the first case.

$\mathcal{V}$ fresh means, that for every given set $\mathcal{H}$ (that does not depend on the algorithm result) we can choose $\mathcal{H} \cap \mathcal{V} = \emptyset$.

Proof We proceed by simultaneous induction on the three statements. The induction is on the size of the last argument to W.

1. Statement 1
ASSUME: 1. $\psi\Gamma \vdash x : \psi\tau$
2. $\phi \leq^{\chi}_{C(\mathcal{U})} \psi$
PROVE: 1. $W_x(\phi, \Gamma, x) = (\mathcal{V}, \tau')$
2. $\phi \leq^{\chi'}_{C(\mathcal{V})} \chi$
3. $\chi'\tau' = \chi\tau$

1.1. Write $\Gamma(x) = \forall\overline{\alpha}.inst(\Gamma(x))$, $\overline{\alpha} = \mathcal{V}$ do not occur in $\phi, \Gamma, \tau, \chi, \psi, \mathcal{U}$

1.2. $\chi\Gamma(x) = \forall\overline{\alpha}.\chi\tau'$
PROOF: $\chi\Gamma(x) = \chi\phi\Gamma(x) = \chi\phi\forall\overline{\alpha}.inst(\Gamma(x)) = \forall\overline{\alpha}.\chi\phi inst(\Gamma(x)) = \forall\overline{\alpha}.\chi\tau'$

1.3. $\chi\Gamma(x) \subseteq \chi\tau$
PROOF: $\psi\Gamma(x) \subseteq \chi\tau$, substitution with χ and $\chi\psi = \chi$
1.4. $\forall\overline{\alpha}.\chi\tau' \subseteq \chi\tau$
1.5. $[\overline{\tau}/\overline{\alpha}]\chi\tau' = \chi\tau$ with $\chi\overline{\tau} = \overline{\tau}$
PROOF: definition of generic instance
1.6. $\phi \leq^{[\overline{\tau}/\overline{\alpha}]\chi}_{C(\mathcal{V})} \chi$
PROOF: Lemma 10.5
1.7. Q.E.D.

2. Statement 2
ASSUME: 1. $\psi\Gamma \vdash M : \diamond$
2. $\phi \leq^{\chi}_{C(\mathcal{U})} \psi$
PROVE: 1. $W_M(\phi, \Gamma, M) = (\mathcal{V}, \phi')$
2. $\phi' \leq^{\chi'}_{C(\mathcal{V})} \chi$
2.1. Case $x\overline{y}$
2.1.1. $\psi\Gamma \vdash x : \overline{\tau}'' \rightarrow \diamond$ and $\psi\Gamma \vdash \overline{y} : \overline{\tau}''$ and $\psi\overline{\tau}'' = \chi\overline{\tau}''$
PROOF: $(APP)^{-1}$
2.1.2. Choose $(\mathcal{V}', \tau') = W_x(\phi, \Gamma, x)$ with
1. $\phi \leq^{\chi_0}_{C(\mathcal{V}')} \chi$
2. $\chi_0\tau' = \chi\overline{\tau}'' \rightarrow \diamond$
PROOF: induction hypothesis on $\chi\Gamma \vdash x : \chi\overline{\tau}'' \rightarrow \diamond$
2.1.3. $(\mathcal{V}_i, \tau_i) = W_x(\phi, \Gamma, y_i)$ with
1. $\phi \leq^{\chi_i}_{C(\mathcal{V}_i)} \chi_{i-1}$
2. $\chi_i\tau_i = \chi_{i-1}\tau_i''$
PROOF: induction hypothesis on $\chi_{i-1}\Gamma \vdash y_i : \chi_{i-1}\tau_i''$ (Lemma 10.3 gives $\chi_{i-1}\psi = \chi_{i-1}$)
2.1.4. χ_n unifies $\tau' = \overline{\tau} \rightarrow \diamond$
PROOF: $\chi_n\tau' = \chi_n\chi_0\tau' = \chi_n\chi\overline{\tau}'' \rightarrow \diamond = (\ldots, \chi_n\chi_{i-1}\tau_i'', \ldots) \rightarrow \diamond = (\ldots, \chi_n\chi_i\tau_i, \ldots) \rightarrow \diamond = \chi_n\overline{\tau} \rightarrow \diamond$ and $\chi_n\chi_i = \chi_n$ by Lemma 10.3
2.1.5. Choose $(\mathcal{W}, \phi'') = mgu(\tau' = \overline{\tau} \rightarrow \diamond)$ and $\phi' = \phi''\phi \leq^{\chi'}_{C(\mathcal{W})} \chi_n$
PROOF: Lemma 10.1
2.1.6. $\phi' \leq^{\chi'}_{C(\mathcal{V}' \cup \overline{\mathcal{V}} \cup \mathcal{W})} \chi$
PROOF: Lemma 10.3
2.2. Case $M_1 \& M_2$
2.2.1. $\psi\Gamma \vdash M_1 : \diamond$ and $\psi\Gamma \vdash M_2 : \diamond$
2.2.2. Choose $(\mathcal{V}_1, \phi'') = W_M(\phi, \Gamma, M_1)$ with $\phi'' \leq^{\chi_1}_{C(\mathcal{V}_1)} \chi$
PROOF: $\chi\Gamma \vdash M_1 : \diamond$
2.2.3. $(\mathcal{V}_2, \phi') = W_M(\phi'', \Gamma, M_2)$ with $\phi' \leq^{\chi_2}_{C(\mathcal{V}_2)} \chi_1$
PROOF: $\chi_1\Gamma \vdash M_2 : \diamond$
2.2.4. $\phi' \leq^{\chi_2}_{C(\mathcal{V}_1 \cup \mathcal{V}_2)} \chi$
PROOF: Lemma 10.3
2.3. Case **def** $L_1 = M_1, \ldots, L_n = M_n$ **in** M'
2.3.1. Choose $\Delta_i', \Gamma_i', \Gamma' = \bigcup_j \Gamma_j'$
1. $\psi\Gamma \oplus (\Delta_1' \cup \ldots \cup \Delta_n') \vdash L_i = M_i : \Delta_i'$
2. $\psi\Gamma \oplus \Gamma' \vdash \Delta_i'$ **gen** Γ_i'
3. $\psi\Gamma \oplus \Gamma' \vdash M' : \diamond$
PROOF: $(DEF)^{-1}$
2.3.2. $\psi\Gamma \vdash D :_{\text{gen}} \psi\Gamma'$
PROOF: (DEF')
2.3.3. Choose $(\mathcal{X}, \phi'', \Gamma'') = W_D(\phi, \Gamma, L_1 = M_1, \ldots, L_n = M_n)$ with
1. $\phi'' \leq^{\chi_1}_{C(\mathcal{X})} \chi$
2. $\chi_1\Gamma'' \subseteq \chi\Gamma'$.
PROOF: induction hypothesis
2.3.4. $\chi_1\Gamma \oplus \chi_1\Gamma'' \vdash M' : \diamond$
PROOF: 2.3.3;2, 2.3.1;3 substituted with χ_1 each and weakening
2.3.5. $(\mathcal{W}, \phi') = W_M(\phi'', \Gamma \oplus \Gamma'', M')$ with $\phi' \leq^{\chi'}_{C(\mathcal{W})} \chi_1$
PROOF: induction hypothesis
2.3.6. $\phi' \leq^{\chi'}_{C(\mathcal{W} \cup \mathcal{X})} \chi$
PROOF: Lemma 10.3

2.4. Q.E.D.

3. Statement 3

ASSUME: 1. $\psi\Gamma \vdash L_1 = M_1, \ldots, L_n = M_n$:gen $\psi\Gamma'$

2. $\phi \leq^{\chi}_{C(\mathcal{U})} \psi$

PROVE: 1. $(\mathcal{V}, \phi', \Gamma'') = W_D(\phi, \Gamma, L_1 = M_1, \ldots, L_n = M_n)$

2. $\phi' \leq^{\chi'}_{C(\mathcal{V})} \chi$

3. $\chi'\Gamma'' \subseteq \chi\Gamma'$.

3.1. There exist Δ'_i, Γ'_i

1. $\chi\Gamma' = \bigcup_i \Gamma'_i$
2. $\chi\Gamma'_i = \Gamma'_i$
3. $\chi\Delta'_i = \Delta'_i$
4. $\chi\Gamma \oplus \bigcup_i \Delta'_i \vdash L_i = M_i : \Delta'_i$
5. $\chi\Gamma \oplus \bigcup_i \Gamma'_i \vdash \Delta'_i$ **gen** Γ'_i

PROOF: $(DEF')^{-1}$

3.2. $(\mathcal{V}_i, \Delta_i) = denv(L_i)$

3.3. Choose ψ' with $\mathrm{dom}(\psi') = \overline{\mathcal{V}}$ such that $\psi'\Delta_i(x) = \Delta'_i(x)$

PROOF: every variable occurs exactly once and the arities of $\Delta_i(x)$, $\Delta_i(x)$ are the same as those of $\Delta'_i(x)$, $\Delta'_i(x)$ which are in turn the same as $\Gamma'_i(x)$, $\Gamma'_i(x)$ which must be equal

3.4. $\phi \leq^{\psi'\chi}_{C(\overline{\mathcal{V}})} \chi$

PROOF: Lemma 10.5

3.5. $\psi'\chi$ unifies $\bigwedge_{x\in\mathrm{dom}(\Delta_i)\cap\mathrm{dom}(\Delta_j)} \Delta_i(x) = \Delta_j(x)$

PROOF: $\psi'\chi\Delta_i(x) = \psi'\Delta_i(x) = \Delta'_i(x)$

3.6. Choose $(\mathcal{X}, \phi'_0) = mgu(\bigwedge_{x\in\mathrm{dom}(\Delta_i)\cap\mathrm{dom}(\Delta_j)} \Delta_i(x) = \Delta_j(x))$ with $\phi_0 = \phi'_0\phi \leq^{\chi_0}_{C(\mathcal{X})} \psi'\chi$

PROOF: Lemma 10.1

3.7. $\chi_0(\Gamma \oplus \Delta) \vdash L_i = M_i : \chi_0\Delta_i$

PROOF: substitute χ_0

3.8. $\chi_0(\Gamma \oplus \Delta \oplus lenv(\Delta_i, L_i)) \vdash M_i : \diamond$

PROOF: (EQN) only rule to derive this

3.9. Choose $(\mathcal{W}_i, \phi_i) = W_M(\phi_{i-1}, \Gamma \oplus \Delta \oplus lenv(\Delta_i, L_i), M_i)$ with $\phi_i \leq^{\chi_i}_{C(\mathcal{W}_i)} \chi_{i-1}$

3.10. Choose $mono = \bigcup_j mtv(\phi_n\Delta_j, \mathrm{dn}(L_j)) = \bigcup_j mtv(\phi_n\Delta, \mathrm{dn}(L_j))$

PROOF: only variables of $\mathrm{dn}(L_j)$ matter in the domain

3.11. Choose $\chi' = \chi_n$

3.12. $\phi' = \phi_n$ and $\Gamma'' = gen(\phi_n\Gamma, mono, \phi_n\Delta)$

PROOF: this is returned

3.13. $\phi' \leq^{\chi'}_{C(\overline{\mathcal{V}}\cup\mathcal{X}\cup\overline{\mathcal{W}})} \chi$

PROOF: Lemma 10.3

3.14. $\chi'\Gamma \oplus \bigcup_j \chi'\Gamma'_j \vdash \chi'\Delta_k$ **gen** $\chi'\Gamma'_k$ for all k

PROOF: identical to 3.1 because $\mathrm{tv}(\Gamma, \Gamma', \Delta') \subseteq C(\overline{\mathcal{V}} \cup \mathcal{X} \cup \overline{\mathcal{W}})$ and $\chi'\Delta_k = \chi'\Delta'_k$

3.15. $gen(\chi'\Gamma, \bigcup_j mtv(\chi'\Delta_j, \mathrm{dn}(L_j)), \chi'\Delta_i) \subseteq \chi'\Gamma'_i$

PROOF: Lemma 10.7;3

3.16. $\chi' gen(\phi_n\Gamma, mono, \phi_n\Delta_i) \subseteq \chi'\Gamma'_i$

PROOF: Lemma 10.7;1 and transitivity

3.17. $\chi' gen(\phi_n\Gamma, mono, \phi_n\Delta) \subseteq \chi'\Gamma'$

PROOF: definition of gen

3.18. $\chi'\Gamma'' \subseteq \chi'\Gamma' = \chi\Gamma'$

PROOF: $\mathrm{tv}(\Gamma') \cap (\overline{\mathcal{V}} \cup \mathcal{X} \cup \overline{\mathcal{W}}) = \emptyset$

4. Q.E.D.

Corollary 10.13 (Completeness) ψ idempotent $\wedge\, \psi\Gamma \vdash M : \diamond \;\Rightarrow\; W_M(id, \Gamma, M) = \phi$ and $\phi \leq_{\mathrm{tv}(\Gamma)} \psi$.

Proof $id \leq^{\psi}_{C(\emptyset)} \psi$. Choose $(\mathcal{V}, \phi) = W_M(id, \Gamma, M)$ $\phi \leq^{\psi'}_{C(\mathcal{V})} \psi$. Now $\mathrm{tv}(\Gamma) \subseteq C(\mathcal{V})$ implies $\phi \leq^{\psi'}_{\mathrm{tv}(\Gamma)} \psi$