By I.H. we have that $T_0 \vdash_{QSTIN} N_1 \Rightarrow N'_1 : \tau'_1 \rightarrow \tau'_2$, and $T_0 \vdash_{QSTIN} N_2 \Rightarrow N'_2 : \tau'_3$ for some $N_1, N'_1, N_2, N'_2, \tau'_1, \tau'_2, \tau'_3$, with $\tau'_1 = \sigma_{ext}(\tau_1)$, and $\tau'_2 = \sigma_{ext}(\tau_2)$.

We also have that $\sigma(\alpha) \leq \tau'_1$, and $\sigma(\alpha) \leq \tau'_3$, and therefore there exists a $\tau = \tau'_1 \land \tau'_3$, and therefore we can apply the corresponding QSTIN rule and get $T_0 \vdash_{Type} N_1, N_2 \Rightarrow N'_1 \uplus N'_2 : \tau$. 

(2) $T_0, C_1 \vdash_{QSTIN} M_1 : \tau_1, T_0, C_2 \vdash_{QSTIN} M_2 : \tau_2$ with $C = C_1 \cup C_2 \cup \{\tau_1 \rightarrow \tau_2 \leq \tau_1, \tau_2 \leq \tau_1, \tau_1 \leq \tau_2, \tau_2 \leq \tau_2\}$.

By I.H. we have that $T_0 \vdash_{Type} N_1 \Rightarrow N'_1 : \tau'_1 \land \tau'_2$ and $T_0 \vdash_{Type} N_2 \Rightarrow N'_2 : \tau'_2$.

If we have that $\sigma(\tau_1) = \tau_1 \rightarrow \tau_1$ for some $\tau_1$ and $\tau_2$, then this is similar to case (1).

If $\sigma(\tau_1) = \Omega$ then we have that we can apply the corresponding application rule of Type and we get $T_0 \vdash_{Type} N_1, N_2 \Rightarrow (N'_1 \uplus N'_2) : \tau'$. This concludes the proof.

As mentioned above an expression is "typable" using the rules above if the set of constraints $C$ obtained has a solution. We briefly describe a simple algorithm to solve a set of constraints. Full details are not shown due to lack of space. The simple algorithm described here has been implemented.

We say that a constraint is inconsistent if it has no solution. An example of such a constraint is an inequality of the form $\alpha \leq \tau \rightarrow \alpha$ where $\alpha$ is a type variable and $\tau$ is an arbitrary type.

Let $C$ be a set of constraints. The idea is to transform $C$ into a set of constraints $C_1$ such that $\sigma$ is a solution of $C_1$ if and only if it is a solution for $C$. The transformation and the building of $\sigma$ are as follows:

1. For all inequalities of the form $\tau \leq \tau$ make $C = C - \{\tau \leq \tau\}$.
2. For all constraints of the form $\alpha \leq \int \text{int}$ make $\sigma(\alpha) = \text{int}$, and make $C = \sigma(C - \{\alpha \leq \text{int}\})$. If an inconsistent inequality is created stop signaling a type error.
3. For all constraints of the form $\alpha \leq \text{bool}$ make $\sigma(\alpha) = \text{bool}$, and make $C = \sigma(C - \{\alpha \leq \text{bool}\})$. If an inconsistent inequality is created stop signaling a type error.
4. For all constraints of the form $\Omega \leq \alpha$ make $\sigma(\alpha) = \Omega$ and make $C = \sigma(C - \{\Omega \leq \alpha\})$. If an inconsistent inequality is created stop signaling a type error.
5. For all constraints of the form $\alpha \leq \tau_1 \rightarrow \tau_2$ make $\sigma(\alpha) = \alpha_1 \rightarrow \alpha_2$, where $\alpha_1$ and $\alpha_2$ are fresh variables, and make $C = \sigma(C - \{\alpha \leq \tau_1 \rightarrow \tau_2\}) \cup \{\tau_1 \leq \alpha_1, \alpha_2 \leq \tau_2\}$. If no inconsistencies have been introduced goto 1, otherwise stop signaling a type error.
6. For all constraints of the form $\tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2$ make $C = (C - \{\tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2\}) \cup \{\tau'_1 \leq \tau_1, \tau'_2 \leq \tau_2\}$. If no inconsistencies were introduced then goto 1, otherwise stop signaling a type mismatch.

The process ends when no more changes are done to the set of constraints. At the end $C$ has been converted into a set of constraints $C_2$ of the form $\tau_1 \leq \tau_2$, where $\tau_1$ and $\tau_2$ are type variables, of the form $\tau_1 \rightarrow \tau_2 \leq \tau_3$, where $\tau_1$ and $\tau_2$ are type variables, or $\tau_1 \leq \tau_2$, where $\tau_1$ and $\tau_2$ are ground types.

We also have a substitution $\sigma$, which will be extended to a solution of $C$ as follows:

For each variable $\alpha$ that occurs in $C$, if there are constraints $\tau_1 \leq \alpha$ and $\tau_2 \leq \alpha$, with $\tau_1 \neq \tau_2$, then make $\sigma(\alpha) = \Omega$, and $C_1 = \sigma(C_1)$.

It is easy to check that the resulting $\sigma$ is a solution to the set of constraints, and the details are left out because of space constraints. As an example, the algorithm described here gives as a solution for example 4 a substitution $\sigma$ such that $\sigma(\alpha_1) = \Omega$ and $\sigma(\alpha_2) = \Omega$.

4 Conclusions and Future Work

We presented here a simple algorithm to infer, in the context of Quasi-static typing, so called Partial Types, and we did so in the context of statically typed languages. This is an improvement over [12], where only type checking is done.

The system presented here does not give the best possible type for a given program expression. The problem is in the algorithm to solve the set of constraints. We are working now on a second algorithm that would give a "better" type. For example, for the expression in example 4, the type $\lambda x : \text{int} \rightarrow \text{int} : \tau \rightarrow \text{int}$ resulting from the algorithm presented here is

$\lambda x : \text{int} \rightarrow \text{int} : \Omega$.

A preliminary version of a more sophisticated algorithm to solve constraints gives us

$\lambda x : \text{int} \rightarrow \text{int} : \Omega$.

which is in a sense, more useful.

One of the problems to integrate this into a language with polymorphic types is the fact that the above rules do not have a most general type property that some typing systems, like ML, enjoy. This means that given an expression, there is no type from which all other possible types for that expression can be derived in some fashion. As an example consider the following expression (in the expression we make use of lists, not included in the original language, we do this because this example has been used elsewhere to illustrate the problem, see [6] and [12]):

$M = \lambda x. \text{cons}(1, x)$

The type of $M$ is $\text{int-list} \rightarrow \text{int-list}$. But if we apply $M$ to a boolean, say $M \text{true}$, then $M$ could be modified as follows by the above system:

$\lambda x : \text{list} \rightarrow \text{list} : \text{cons}(1, x)$

with type $\Omega \rightarrow \text{list}$. One could conclude that the type $\forall x. \text{list} \rightarrow \text{list}$ is a valid type for $M$, but the type $\text{bool-list} \rightarrow \text{bool-list}$ is not a valid type for $M$. 

34