A *More* Direct Algorithm for Type Inference in the Rank-2 Fragment of the Second-Order λ-Calculus

(Revised July 27, 2007)

Brad Lushman
University of Waterloo
200 University Ave. W.
Waterloo, Ontario, Canada
bmlushma@uwaterloo.ca

Gordon V. Cormack
University of Waterloo
200 University Ave. W.
Waterloo, Ontario, Canada
gvcormac@uwaterloo.ca

ABSTRACT

We present an algorithm for rank-2 type inference in the second-order λ-calculus. Our algorithm differs from the well-known algorithm of Kfoury and Wells in that it employs only a quadratically fewer type variables and inequalities. Our algorithm consists of a translation from a λ-term to an instance of R-ASUP (a decidable superset of ASUP) in which the variables correspond more directly to features in the original term. We claim that our construction, being simpler and more direct, is more amenable to proof and extension.

1. INTRODUCTION

Type inference for the second-order λ-calculus was one of the most important open problems in programming language theory, until the question was answered negatively by Wells [11]. Even after Wells' result, research in polymorphic type inference has continued. An important approximate solution is that of Kfoury and Wells, which we call Algorithm KW. Their algorithm answers the type inference problem for the rank-2 fragment of the second-order λ-calculus [6]. Within this sublanguage, a function may require a polymorphic argument, but no function may demand an argument that is itself a function demanding a polymorphic argument. In other words, the $\forall$-symbol, which denotes polymorphism, may be nested to the left of at most one $\rightarrow$ symbol.

We also know from Wells' result that type inference is undecidable at all higher ranks. However, rank-2 constructions subsume the kind of polymorphic abstractions permitted by `let`-polymorphism in ML-like languages; hence, being able to express widely used functional programming patterns, they form an important sublanguage of the second-order λ-calculus. A rank-2 inference algorithm would allow the programmer to rewrite `let`-constructions as the application of functions to arguments, thereby facilitating extensions supporting separate compilation of a polymorphic abstraction from its argument.

Despite these advantages of rank-2 inference, it has not found its way into the more popular typed functional languages. We believe the absence of rank-2 inference is due in part to certain shortcomings of Algorithm KW. Two of these shortcomings are the scheme by which it represents subexpressions with type variables, and the complex source-level transformation (known as θ-reduction) that it employs before the type inference process begins. Both of these shortcomings make the algorithm difficult to understand and trace; the latter in particular makes it hard to translate a failed inference into a type error that the programmer can understand, because the error that arises was found in the transformed program instead of the original program.

We present in this paper an algorithm for type inference in the rank-2 fragment of the second-order λ-calculus that addresses the first of these shortcomings. The result is an algorithm that is much clearer to state. Our results are based on properties of the *R-acyclic semiunification problem* (a generalization of the acyclic semiunification problem of Kfoury, Tiuryn, and Urzyczyn [5]), whose properties we state and prove elsewhere [7].

The remainder of this paper is organized as follows. In Section 2, we introduce the semiunification problem, its subproblems, and the properties of the R-acyclic semiunification problem that we will need. Section 3 contains a summary of our algorithm. Section 4 summarizes Algorithm KW, highlighting the differences between that algorithm and ours. In Section 5 we show that our algorithm computes the same answers as Algorithm KW. We summarize and discuss our results in Section 6.

2. BACKGROUND AND RELATED WORK

The success of ML and its descendents is in a large part due to the polymorphic type assignment algorithm W of Damas and Milner [1]. Since then, language designers have sought type inference algorithms for richer type systems. Of particular interest was the type inference problem for the second-order λ-calculus of Girard [2] and Reynolds [9]. This problem remained open for decades before it was resolved negatively by Wells [11]. At the same time, type inference was proved undecidable for all rank restrictions of the second-order λ-calculus beyond 2. On the other hand, the rank-2 fragment itself, which subsumes, for example, the `let`-polymorphism of ML, was shown decidable, via Algorithm KW [6]. However, Algorithm KW remains largely a theoretical result; our work in this paper removes some of the shortcomings, as mentioned in Section 1, that hinder its general use.

At the heart of many type inference systems is the unification problem of Robinson [10]. In richer type systems, such as that of the second-order λ-calculus, richer unification-related problems arise. Of particular interest here is the semiunification problem (SUP) and its descendents. SUP was introduced by Henglein [3] as part of a study of polymorphic recursion in the Milner-Mycroft calculus [8]. Though SUP was initially believed to be solvable, it was proved undecidable by Kfoury, Tiuryn, and Urzyczyn [4]. SUP's undecidability was used in Wells' proof of the undecidability of type inference for the second-order λ-calculus [11]. On the other hand, the decidability of a subproblem, the *acyclic* semiunification problem (ASUP, defined in [5]), forms the basis for Algorithm KW. More recently, a larger and more natural decidable subproblem, known as the *R-acyclic* semiunification problem (R-ASUP), is presented in [7], and underlies our present work.

2.1 SUP, ASUP, and R-ASUP

The algorithms presented in this paper are based on the semiunification problem (SUP) and its subproblems. In this section, we present the basic definitions and results that we will need in the remainder of the paper.

DEFINITION 1 (SUP). *Given a term algebra $\mathbb{A}$, an instance of SUP is a set Γ of pairs (called* inequalities*) $\{\tau_i \leq \mu_i\}_{i=1}^N$ where $\tau_i, \mu_i \in \mathbb{A}$ for all i. A solution of Γ is a substitution S such that there exist substitutions $S_1, \ldots, S_N$ such that*

$$\begin{aligned} \tau_1 S S_1 &= \mu_1 S \\ \tau_2 S S_2 &= \mu_2 S \\ &\cdots \\ \tau_N S S_N &= \mu_N S \end{aligned}$$

Although SUP itself is undecidable, it has two notable decidable subsets. The first is the *acyclic semiunification problem* (ASUP), which forms the basis for the algorithm KW. ASUP is defined in [5]; a solution procedure is given in [6], and reproduced in Section 3.4. The second is the *R-acyclic semiunification problem* (R-ASUP), which forms the basis for our algorithm. R-ASUP is presented in detail in [7]; we summarize its properties here.

DEFINITION 2. *Given a SUP instance Γ, the* graph of Γ*, denoted $G(\Gamma)$, is a directed graph, defined as follows:*

- *the vertices in $G(\Gamma)$ are the inequalities $\tau \leq \mu$.*
- *given vertices (inequalities) $\tau_i \leq \mu_i$ and $\tau_j \leq \mu_j$, there is an edge from $\tau_i \leq \mu_i$ to $\tau_j \leq \mu_j$ iff*

$$\mathrm{Vars}(\mu_i) \cap \mathrm{Vars}(\tau_j) \neq \emptyset \ .$$

For any expression τ, $\mathrm{Vars}(\tau)$ represents the set of variables in τ.

DEFINITION 3. *Let $G = G(\Gamma)$ be a SUP graph. Define relations R and R' on variables in G as follows:*

- *for variables α and β, $\alpha R \beta$ if there are vertices $\tau_i \leq \mu_i$ and $\tau_j \leq \mu_j$ such that there is a directed path from $\tau_i \leq \mu_i$ to $\tau_j \leq \mu_j$, and $\alpha \in \mathrm{Vars}(\mu_i)$ and $\beta \in \mathrm{Vars}(\mu_j)$.*
- *for variables α and β, $\alpha R' \beta$ if there are vertices $\tau_i \leq \mu_i$ and $\tau_j \leq \mu_j$ such that there is a directed path* of nonzero length *from $\tau_i \leq \mu_i$ to $\tau_j \leq \mu_j$, and $\alpha \in \mathrm{Vars}(\mu_i)$ and $\beta \in \mathrm{Vars}(\mu_j)$.*

Note that R' is a subrelation of R.

DEFINITION 4 (R-ACYCLIC; R-ASUP). *A SUP graph G is R-acyclic if for all variables α, β, such that $\alpha R' \beta$, we have $\neg(\beta R^+ \alpha)$, where R^+ is the transitive closure of R. R-ASUP is the restriction of SUP to instances whose graphs are R-acyclic.*

The following properties of R-ASUP, which we use in the remainder of this paper, are proved in [7]:

THEOREM 1. *R-ASUP is a decidable subset of SUP; moreover, it is decidable by the same procedure that decides ASUP.*

THEOREM 2. *R-ASUP is a strict superset of ASUP.*

3. ALGORITHM LC

We present in this section our algorithm for rank-2 type inference. We will contrast our algorithm with Algorithm KW in the Section 4.

3.1 λ-Labelling

We first indicate which abstractions are allowed to be polymorphic, and which must be monomorphic, by labelling each abstraction in the term with either 1, 2, or 3, as follows:

- a λ^1-abstraction is any abstraction whose argument has been supplied;
- a λ^3-abstraction is an abstraction whose argument has not been supplied, but itself appears as a (not necessarily proper) subexpression of some function argument;
- a λ^2-abstraction is any other abstraction.

The λ^1- and λ^2-abstractions are polymorphic abstractions, and the λ^3-abstractions are monomorphic abstractions.

We begin by determining the *active variables* in the term. These are the bound variables that are not matched with arguments[1]:

$$\begin{aligned} \mathrm{act}(x) &= [] \text{ (the empty list)} \\ \mathrm{act}(\lambda x.M) &= x :: \mathrm{act}(M) \\ \mathrm{act}(MN) &= \begin{cases} [] & \text{if } \mathrm{act}(M) = [], \\ x' & \text{if } \mathrm{act}(M) = x :: x' \end{cases} \end{aligned}$$

For a given term M, the labelled version of M, denoted M^λ, is given by

$$M^\lambda = \mathrm{lbl}(M, \mathrm{act}(M), 2) \ ,$$

where the function "lbl" is defined by

$$\begin{aligned} \mathrm{lbl}(x, X, i) &= x \\ \mathrm{lbl}(\lambda x.M, X, i) &= \begin{cases} (\lambda^i x.\mathrm{lbl}(M, X, i)) & \text{if } x \in X, \\ (\lambda^1 x.\mathrm{lbl}(M, X, i)) & \text{if } x \notin X. \end{cases} \\ \mathrm{lbl}((MN), X, i) &= \mathrm{lbl}(M, X, i)\mathrm{lbl}(N, \mathrm{act}(N), 3) \end{aligned}$$

3.2 θ-Reduction

Before translating from a λ-labelled expression to SUP, we follow the example of KW, and first transform the expression, via a process known as *θ-reduction*. The process of θ-reduction consists of applying the following four reduction rules, until it is no longer possible to do so:

- (θ_1) $((\lambda^1 y.N)P)Q \to (\lambda^1 y.NQ)P$
- (θ_2) $\lambda^3 z.(\lambda^1 y.N)P \to (\lambda^1 v.\lambda^3 z.(N'))(\lambda^3 w.(P'))$, where $N' = N[vz/y]$, $P' = P[w/z]$, and v and w are fresh variables
- (θ_3) $N((\lambda^1 y.P)Q) \to (\lambda^1 y.NP)Q$
- (θ_4) $(\lambda^1 y.(\lambda^2 x.N))P \to \lambda^2 x.((\lambda^1 y.N)P)$

[1]We implicitly assume that all bound variables have been named distinctly from each other and from all free variables.

An expression in which no θ-reduction is possible is said to be in *θ-normal form*. Terms in θ-normal form are guaranteed to have the following shape:

$$\lambda^2 .x_1 \cdots x_m .(\lambda^1 y_1 \cdots ((\lambda^1 y_n .E_n)E_{n-1}) \cdots)E_0 \ ,$$

in which each E_i may contain λ^3-abstractions, but no λ^1- or λ^2-abstractions.

Because θ-normal forms have such a rigid structure, it is easier to reason about them than about arbitrary expressions. However, the degree of structure in θ-normal forms also implies that the amount of change that a term must undergo to reach θ-normal form can be drastic; a programmer might not even recognize a θ-normal equivalent of a program he wrote. Further, as translation to ASUP occurs on the θ-normal form, any type errors are detected in relation to the θ-normal form, not the original term; hence, θ-reduction hinders our ability to communicate type errors back to the programmer in a meaningful way.

For these reasons, a practical inference algorithm must abandon the structure provided by θ-normal forms in favour of a more general solution. We do not address this need here; we leave it for future work.

3.3 Translation to R-ASUP

After labelling and θ-reduction, the term has the form

$$\lambda^2 .x_1 \cdots x_m .(\lambda^1 y_1 \cdots ((\lambda^1 y_n .E_n)E_{n-1}) \cdots)E_0 \ ,$$

in which each E_i may contain λ^3-abstractions, but no λ^1- or λ^2-abstractions. We then translate the term into an instance of SUP. The resulting instance uses the following variables:

- for each λ^2-bound variable x_i, the variable β_{x_i} represents its specializable type
- for each λ^1-bound variable y_i, the variable β_{y_i} represents its specializable type
- for each λ^3-bound variable z_i, the variable γ_{z_i} represents its non-specializable type
- for each free variable w_i, the variable β_{w_i} represents its specializable type
- for each subexpression E of M, δ_E represents its derived type

We then translate the term M to a set of SUP equalities and inequalities as follows—for each subexpression E_k:

- For each abstraction $\lambda^3 z_i.N$, include the equality $\delta_{\lambda^3 z_i.N} = \gamma_{z_i} \to \delta_N$.
- For each occurrence x_{ij} of a variable x_i, include the inequality $\beta_{x_i} \leq \delta_{x_{ij}}$.
- For each occurrence y_{ij} of a variable y_i, include the inequality $\beta_{y_i} \leq \delta_{y_{ij}}$.

- For each occurrence w_{ij} of a free variable w_i, include the inequality $\beta_{w_i} \leq \delta_{w_{ij}}$.
- For each occurrence z_{ij} of a variable z_i, include the equality $\gamma_{z_i} = \delta_{z_i j}$.
- For each application MN, include the equality $\delta_M = \delta_N \to \delta_{MN}$.
- If there is a user-supplied type annotation $\forall\vec{\alpha}.\tau$ for some λ^2-bound variable x_i, include the equality $\beta_{x_i} = \tau$.
- For each free variable w_i, if a type environment supplies w_i with the type $\forall\vec{\alpha}.\tau$, include the equality $\beta_{w_i} = \tau$.

Each equality $\tau = \mu$ is merely syntactic sugar for the inequality $\alpha \to \alpha \leq \tau \to \mu$, where α is a fresh variable. Finally, for each redex $(\lambda^1 y_j.E_{j+1})E_j$, we add the equality $\beta_{y_j} = \delta_{E_j}$.

For proof that the resulting instance belongs to R-ASUP, see later in this paper.

3.4 R-ASUP Solution and Type Recovery

To compute the type of a term M, we solve the R-ASUP instance we computed in the previous section. The R-ASUP solution procedure is identical to the ASUP solution procedure presented in [6]. It is a "redex procedure", in which reducible expressions (*redexes*) are repeatedly found and reduced. There are two kinds of redex:

- *redex-I*: Let $\tau \leq \mu$ be an inequality. Let α be a variable and τ_1 be a compound expression such that α and τ_1 occur at corresponding positions within μ and τ, respectively. Then a redex-I is said to exist and reduction of the redex-I consists of replacing α throughout the problem instance by τ_1', which is τ_1 with all variables replaced by fresh ones.
- *redex-II*: Let $\tau \leq \mu$ be an inequality. Let α be a variable and μ_1 and μ_2 be expressions such that $\mu_1 \neq \mu_2$, α and μ_1 occur at corresponding positions within τ and μ, respectively, and α and μ_2 occur at corresponding positions within τ and μ, respectively. Then a redex-II is said to exist and reduction of the redex-II consists of applying the most general unifier of μ_1 and μ_2 throughout the instance (according to Robinson's unification algorithm [10]) and failing if unification fails.

At the end of the redex procedure, when no redexes remain, the accumulated substitution performed, S, is the solution of the R-ASUP instance.

To recover a term's type from the associated ASUP instance, KW begins by solving the instance, via the same redex procedure as we presented in Section 3.4. If S is the solution of the instance, then the final type is given by

$$\forall.(\forall.\tau_1 \to \cdots \to \forall.\tau_m \to \delta_{E_{n+1}}S) \ ,$$

where $\forall.\tau_i$ is the user-annotated type for the λ^2-variable x_i, or $\forall\alpha.\alpha$ if none was supplied.

4. ALGORITHM KW

We present in this section the original rank-2 typability algorithm for the second-order λ-calculus. This algorithm appears in [6]; our purpose here is to contrast it with our own algorithm.

4.1 Translation to ASUP

The abstraction-labelling and θ-reduction steps for KW are the same as for LC. The difference lies in the translation procedure, which targets ASUP for KW, whereas LC targets R-ASUP. As we noted previously, the input is always a θ-normal form:

$$\lambda^2 x_1 \cdots x_m.(\lambda^1 y_1.(\lambda^1 y_2.(\cdots((\lambda^1 y_n.E_{n+1})E_n)\cdots))E_2)E_1,$$

with free variables $w_1, \ldots w_p$. Each E_i may introduce variable bindings; the variables bound in E_i will be called $z_{i,1}, z_{i,2}, \ldots$.

The corresponding instance of ASUP contains the following variables:

- For $1 \leq j \leq m$, $1 \leq i \leq n+1$, the variable $\beta^x_{i-1,j}$, representing the type of the parameter x_j (which may be specialized) within the subexpression E_i.
- For $1 \leq j \leq p$, $1 \leq i \leq n+1$, the variable $\beta^w_{i-1,j}$, representing the type of the free variable w_j (which may be specialized) within the subexpression E_i.
- For $1 \leq j \leq n$, $j < i \leq n+1$, the variable $\beta^y_{i-1,j}$, representing the type of the parameter y_j (which may be specialized) within the subexpression E_i.
- For each bound variable $z_{i,j}$, the variable $\gamma_{i,j}$, representing its (non-specializable) type.
- For each subexpression M of each E_i, the variable δ_M, representing the type derived for M (different occurrences of the same subexpression are considered distinct and are assigned different δ-variables).

Then, for each subexpression N of each E_i, the ASUP instance contains the following inequalities:

- If $N = x_j$, the inequality $\beta^x_{i-1,j} \leq \delta_N$.
- If $N = w_j$, the inequality $\beta^w_{i-1,j} \leq \delta_N$.
- If $N = y_j$, the inequality $\beta^y_{i-1,j} \leq \delta_N$.
- If $N = z_{i,j}$, the equality $\gamma_{i,j} = \delta_N$.
- If $N = PQ$, the equality $\delta_P = \delta_Q \to \delta_N$.

- If $N = \lambda z_{i,j}.P$, the equality $\gamma_{i,j} \to \delta_P = \delta_N$.

As before, an equality $\tau = \mu$ is syntactic sugar for the inequality $\alpha \to \alpha \leq \tau \to \mu$, where α is a fresh variable.

In addition, the instance contains the following inequalities:

- For each β-redex $(\lambda y_i.P_i)E_i$, include the equality $\beta^y_{i,i} = \delta_{E_i}$.
- For each x_j, $1 \leq i \leq n$, include the inequality $\beta^x_{i-1,j} \leq \beta^x_{i,j}$.
- For each y_j, $j < i \leq n$, include the inequality $\beta^y_{i-1,j} \leq \beta^y_{i,j}$.
- For each w_j, $1 \leq i \leq n$, include the inequality $\beta^w_{i-1,j} \leq \beta^w_{i,j}$.

Finally, for each of the free variables $w_1, \ldots w_p$, we consult a type environment A (if one has been supplied) for typing information. If, for a given w_i, $A(w_i) = \forall\vec{\alpha}.\sigma$ for some rank 0 type σ, we include the equality $\beta^w_{0,i} = \sigma$ in the ASUP instance. Similarly, if the variable x_j is supplied, via programmer annotation, the type $\forall\vec{\alpha}.\sigma$ for some rank 0 type σ, then we include the equality $\beta^x_{0,j} = \sigma$ in the problem instance.

A few properties of this translation are worth noting. First, each variable x_j, y_j, and w_j gets, respectively, $n+1$, $n-j+1$, and $n+1$ specializable type variables to track its type, one for each expression E_i in which the variable could possibly occur. These variables are called, respectively, $\beta^x_{0,j}, \ldots, \beta^x_{n,j}$, $\beta^y_{j,j}, \ldots, \beta^y_{n,j}$, and $\beta^w_{0,j}, \ldots, \beta^w_{n,j}$. Second, the variables associated with x_j, y_j, and w_j, are related to one another by chains of inequalities: $\beta^x_{0,j} \leq \cdots \leq \beta^x_{n,j}$, $\beta^y_{j,j} \leq \cdots \leq \beta^y_{n,j}$, and $\beta^w_{0,j} \leq \cdots \leq \beta^w_{n,j}$. By contrast, our translation does not produce these chains of inequalities; further, it employs a *single* variable for each of x_j, y_j, and w_j.

4.2 Type Recovery

The type recovery step for KW is the same as for LC.

In summary, the important difference between our algorithm and KW is that we use only one type variable for each abstraction parameter and free variable. In the next section, we show that our algorithm produces equivalent results to KW.

5. REDUCING THE NUMBER OF SUP VARIABLES

In this section we show that each variable in the term to be typed can be represented by a *single* type variable.

Recall that under the KW translation to ASUP, each variable x_j, y_j, and w_j in a θ-normal term E gets, respectively, $n+1$, $n-j+1$, and $n+1$ specializable type variables to track its type, one for each expression E_i in which the variable could possibly occur. These variables are called, respectively, $\beta^x_{1,j}, \ldots, \beta^x_{n+1,j}$, $\beta^y_{j+1,j}, \ldots, \beta^y_{n+1,j}$, and $\beta^w_{1,j}, \ldots, \beta^w_{n+1,j}$. Second, the variables associated with x_j, y_j, and w_j, are related to one another by chains of inequalities: $\beta^x_{1,j} \leq \cdots \leq \beta^x_{n+1,j}$, $\beta^y_{j+1,j} \leq \cdots \leq \beta^y_{n+1,j}$, and $\beta^w_{1,j} \leq \cdots \leq \beta^w_{n+1,j}$. Hence, any information we deduce for a variable "lower" in the chain will get propagated, via redex-I reduction, "up" the chain to the remaining variables. Collectively, we shall refer to the variables $\beta^x_{i,j}$, $\beta^y_{i,j}$, and $\beta^w_{i,j}$ as *β-variables*. Further, we write $\beta^{\cdot}_{i,j}$ when the distinction among x, y, and w is unimportant. We use the notation $\beta^{\cdot}_{\perp,j}$ to stand for $\beta^x_{1,j}$, $\beta^w_{1,j}$, or $\beta^y_{j+1,j}$.

Next, we observe a few properties of the ASUP solution procedure (which is the same as the R-ASUP solution procedure) in relation to the placement of variables:

OBSERVATION 1. *If a variable α is replaced during redex reduction, then it must occur as part of a redex—this occurrence is always on the righthand side of some inequality.*

Observation 1 is immediate from the definition of the redex algorithm.

OBSERVATION 2. *For $i > \perp$, $\beta^{\cdot}_{i,j}$ only occurs on the lefthand sides of inequalities, except within the inequality $\beta^{\cdot}_{i-1,j} \leq \beta^{\cdot}_{i,j}$.*

Observation 2 is also immediate.

Since $\beta^{\cdot}_{i,j}$, $i > 1$, only occurs on the righthand side of an inequality when it occurs alone, it cannot occur as part of a redex-II. Hence, it can only be replaced as part of a redex-I reduction. This, in turn, can only take place if the variable $\beta^{\cdot}_{i-1,j}$ is replaced during redex reduction. By induction, for all $i > 1$, the variable $\beta^{\cdot}_{i,j}$ can only be replaced if the variable $\beta^{\cdot}_{\perp,j}$ is replaced.

Each $\beta^{\cdot}_{\perp,j}$ can be replaced in only one way:

- $\beta^x_{1,j}$ is replaced via the equality $\beta^x_{1,j} = \tau$ (i.e., the inequality $\alpha \to \alpha \leq \beta^x_{1,j} \to \tau$) if the parameter x_j is annotated with $\forall.\tau$, where τ is a rank 0 type.
- $\beta^w_{1,j}$ is replaced via the equality $\beta^w_{0,j} = \tau$ (i.e., the inequality $\alpha \to \alpha \leq \beta^w_{0,j} \to \tau$) if the initial type environment associates the free variable w_j with the type $\forall.\tau$, where τ is a rank 0 type.
- $\beta^y_{j+1,j}$ is replaced via the equality $\beta^y_{j+1,j} = \delta_{E_j}$ (i.e., the inequality $\alpha \to \alpha \leq \beta^y_{j+1,j} \to \delta_{E_j}$) because of the redex $(\lambda y_j.P_j)E_j$.

These are the only inequalities in which the variables $\beta^{\cdot}_{\perp,j}$ occur on the righthand side.

Assuming these replacements occur (in the case of $\beta^y_{j+1,j}$, they will surely occur), and assuming they result in the

replacement of $\beta^{\vec{}}_{\vec{1},j}$ by something other than just another variable, then there will be a redex-I in the inequality $\beta^{\vec{}}_{\vec{1},j} \leq \beta^{\vec{}}_{\vec{1}+1,j}$. Suppose that $\beta^{\vec{}}_{\vec{1},j}$ has been replaced by an expression $\tau^{\vec{}}_j$. Then redex reduction replaces $\beta^{\vec{}}_{\vec{1}+1,j}$ by the expression $\tau^{\vec{}\prime}_j$, which is $\tau^{\vec{}}_j$, with all variables renamed consistently to fresh variables. Similarly, $\beta^{\vec{}}_{\vec{1}+2,j}$ is replaced by $\tau^{\vec{}\prime\prime}_j$, and so on. Therefore, we have proved the following:

THEOREM 3. *Given j and the choice of x, y, of w, if $\beta^{\vec{}}_{\vec{1},j}$ is replaced by an expression $\tau^{\vec{}}_j$, then all $\beta^{\vec{}}_{i,j}$ are replaced by expressions that differ from $\tau^{\vec{}}_j$ by consistent renaming of variables to fresh ones. In particular, they are all structurally equivalent.*

The replacement of variables by fresh ones is important, because it keeps the problem instance within the realm of ASUP—for each i, the variable $\beta^{\vec{}}_{i,j}$ occurs within the set V_i and all sets V_i are declared by ASUP to be pairwise disjoint. Therefore, to remain within ASUP, the variables in each $\tau^{\vec{}(i)}_j$ must be pairwise disjoint sets.

In [7], we define the notion of a *solved inequality*:

DEFINITION 5 (SOLVED). *An inequality $\tau \leq \mu$ is* solved of order 0 *if there is a substitution S such that $\tau S = \mu$, and the variables of τ do not occur on the right-hand side of any inequality in the instance. An inequality $\tau \leq \mu$ is* solved of order k *for $k > 0$ if there is a substitution S such that $\tau S = \mu$, and the variables of t do not occur on the right-hand side of any inequality in the instance that is not solved of order j for some $j < k$. An inequality is* solved *if it is solved of order k for some $k \geq 0$.*

Informally, $\tau \leq \mu$ is solved if $\tau S = \mu$ for some S, and the variables of τ do not occur on the righthand side of any unsolved inequality in the instance. The important property of solved inequalities is that they will never contain a redex:

THEOREM 4. *If an inequality is solved, then the redex algorithm will never find a redex in it.*

This result is proved in [7]. The theorem allows us to make the following observation about chains of β-variables:

OBSERVATION 3. *Once the redex-I's in the chains of β-variable inequalities have been reduced, the chains are solved.*

Therefore, once the variables $\beta^{\vec{}}_{i,j}$ have been replaced by $\tau^{\vec{}(i)}_j$, we may assume that the chains of β-variable inequalities are no longer there. Therefore, for all practical purposes, the expressions $\tau^{\vec{}(i)}_j$ only occur on left-hand sides.

OBSERVATION 4. *If a variable only occurs on left-hand sides of inequalities in a SUP instance, then it will never be replaced by the ASUP solution algorithm.*

This is clear—since redex reductions arise from replacements on the righthand side of an inequality, if a variable never occurs on a righthand side, it will never be part of a redex.

Depending on the instance, we sometimes have a choice regarding whether a particular variable replacement is performed by one or more of the substitutions $S_1, \ldots, S_n$, or by the solution S. We prefer solutions in which S does as little work as possible; we formalize this notion below:

DEFINITION 6. *Let $\Gamma = \{\tau_i \leq \mu_i\}_{i=1}^n$ be a SUP instance with solution S. S is called* canonical *if*

$$\text{dom } S \subseteq \bigcup_{i=1}^{n} \text{Vars}(\mu_i) \ .$$

The following result shows that any SUP solution has a "canonical core" that is also a solution:

THEOREM 5. *If a SUP instance Γ has a solution S, then S can be written as $S'_C \circ S_C$, where S_C is canonical,* dom $S_C \cap$ dom $S'_C = \emptyset$, *and S_C solves Γ.*

PROOF. For each inequality $\tau_i \leq \mu_i$ in Γ, there is a substitution S_i such that $\tau_i S S_i = \mu S$. Let

$$V = \bigcup_{i=1}^{n} \text{Vars}(\mu_i) \ .$$

Let $S_C = S|_V$ (the restriction of S to variables in V) and $S'_C = S \setminus S_C$ (i.e., the restriction of S to variables not in V). Then S_C is canonical by construction, $S = S'_C \circ S_C$, and dom $S_C \cap$ dom $S'_C = \emptyset$. It remains to show that S_C solves Γ. We have

$$\tau_i S S_i = \mu S \ .$$

Hence, since S_C and S'_C commute (their domains are disjoint),

$$\tau_i S_C S'_C S_i = \mu S'_C S_C \ .$$

Since dom $S'_C \cap V = \emptyset$, we have

$$\tau_i S_C S'_C S_i = \mu S_C \ .$$

Grouping S'_C and S_i together as $S_i \circ S'_C$, we see that S_C solves each inequality in Γ; hence, it solves Γ. □

From now on we implicitly assume that solutions of SUP instances are canonical. Note that any solution computed by the redex procedure arises from redexes whose reduction produces substitutions on at least one right-hand side; hence the redex procedure always computes canonical solutions.

THEOREM 6. *Let $\tau \leq \mu$ be an inequality in a SUP instance Γ, such that the variables in τ do not occur on any righthand sides. Let τ' be a consistent variable renaming of τ, such that the variables in τ' also do not occur on any righthand sides. Let $\Gamma' = (\Gamma \setminus \{\tau \leq \mu\}) \cup \{\tau' \leq \mu\}$. Then Γ' has a solution if and only if Γ has a solution. Moreover, Γ and Γ' have the same solutions.*

PROOF. There exist renaming substitutions ρ and ρ' such that $\tau\rho = \tau'$ and $\tau'\rho' = \tau$. If S solves Γ, then there is a substitution S_1 such that $\tau S S_1 = \mu S$. But then $\tau'\rho' S S_1 = \mu S$. Taking $S' = \rho' \circ S$, and noting that the variables in the domain of ρ' do not intersect those in the domain of S (because S is assumed canonical), we have $\rho' \circ S = S \circ \rho'$. Thus, $\tau' S \rho' S_1 = \mu S$. Letting $S'_1 = S_1 \circ \rho'$, we have $\tau' S S'_1 = \mu S$. As S still solves the remaining inequalities in Γ, it follows that S solves Γ'. Similarly, any solution of Γ' also solves Γ. □

The expressions $\tau_j^{(i)}$ that replace the variables $\beta_{i,j}$ contain only variables that appear on the lefthand sides of inequalities (after the chains of β-variables, and other solved inequalities, have been removed). Hence, we may freely rename the variables in $\tau_j^{(i)}$, so long as the variable names we choose also do not occur on any right-hand sides. Thus we can choose to rename variables such that $\tau_j^{(i)} = \tau_j^{(k)}$ for all i, k, without affecting the solution. What we obtain is the same instance that would result if we used a single variable for each of x_j, y_j, w_j. Therefore, we have the following:

THEOREM 7. *Consider an expression of the form*

$$\lambda x_1 \cdots x_m.(\lambda y_1.(\lambda y_2.(\cdots((\lambda y_n.E_{n+1})E_n)\cdots))E_2)E_1,$$

with free variables $w_1, \ldots w_p$, with all bound variables uniquely named, distinctly from all free variables, and in which each E_i may introduce variable bindings, called called $z_{i,1}, z_{i,2}, \ldots$. Let Γ be the ASUP instance derived for this expression, as described before. Let Γ' be the SUP instance produced by the Algorithm LC, i.e., in which each term variable is represented by exactly one SUP variable. Then Γ and Γ' have the same solutions.

The theorem establishes that the SUP instance obtained by our new translation procedure has the same solution as the corresponding ASUP instance. However, the theorem does not guarantee that, with the instance in this form, the redex procedure will be able to find the solution, as the output of this translation procedure will, in general, lie outside of ASUP. If, in order to find the solution, a procedure must fill in the variables and inequalities we just removed, then we are no further ahead. However, the following theorem shows that this is not necessary:

THEOREM 8. *Every SUP instance produced by the procedure outlined in Theorem 7 is an instance of R-ASUP.*

PROOF. Let Γ be the ASUP instance obtained from some λ-term M. Since ASUP is contained in R-ASUP, Γ is an R-ASUP instance. Let Γ_1 be the result of removing all of the β-chains from Γ. Since removing nodes from a graph does not create any new paths, Γ_1 is an R-ASUP instance. Now, for each β-variable, there is at most one inequality $\tau_k \leq \mu_k$ that contains it on the right-hand side (moreover, since this inequality is phrased as an equality, it follows that $\tau_k = \alpha_k \to \alpha_k$, where α_k occurs in no other inequality). By merging all of the β-variables pertaining to a particular term variable (assume we rename all $\beta_{\bot,j}$ to β_j), we create edges from $\tau_k \leq \mu_k$ to each inequality that had $\beta_{i,j}$ on the left-hand side. Can any of these new paths violate R-acyclicity? Since the original instance was an ASUP instance, any paths that violate R-acyclicity must contain these new edges. Since $\tau_k = \alpha_k \to \alpha_k$, and α_k is fresh, the vertex $\tau_k \leq \mu_k$ has no in-edges. So any path containing $\tau_k \leq \mu_k$ contains it as the first node in the path. Thus, for R-acyclicity to fail, we need to find a variable ϵ outside μ_k such that $\epsilon R \zeta$ for some variable ζ in μ_k. But since the variables in μ_k (i.e., β_j and the type expression with which it is equated) never occur anywhere else on a right-hand side, it follows that there is no variable ϵ outside μ_k such that $\epsilon R \zeta$ for any $\zeta \in \mathrm{Vars}(\mu_k)$, and therefore the instance is R-acyclic. □

As a consequence of Theorem 8, we can simply apply the redex procedure to the R-ASUP instance obtained via Theorem 7, and be guaranteed of termination with the same answer.

6. DISCUSSION

We present in this paper a direct algorithm for computing rank-2 types for terms in System F. Our algorithm differs from the original KW algorithm by eliminating a quadratic number of variables from the SUP instance associated with the given term. As a consequence, we now produce SUP instances that are linear in size, in proportion to the size of the input, where previously the size of the SUP instance was quadratic. Since the ensuing step, (R-)ASUP solution, can be exponential (ASUP is DEXPTIME-complete), reducing the size of the input to this step may pay dividends in time saved.

We leave to future work the problem of eliminating the θ-reduction step from the type inference process, so that we may produce a truly syntax-directed translation from the typability problem to SUP.

7. REFERENCES

[1] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM Press, 1982.

[2] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972. Summary in *Proceedings of the Second*

Scandinavian Logic Symposium (J.E. Fenstad, editor), North-Holland, 1971 (pp. 63–92).

[3] Fritz Henglein. Semi-unification. Technical Report (SETL Newsletter) 223, New York University, April 1988.

[4] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102:83–101, 1993.

[5] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *Journal of the Association for Computing Machinery*, 41(2):368–398, March 1994.

[6] A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In *1994 ACM Conference on LISP and Functional Programming*, pages 196–207. ACM Press, 1994.

[7] Brad Lushman and Gordon V. Cormack. The R-acyclic semiunification problem. Technical Report CS-2006-06, University of Waterloo, 2006. Available at http://www.cs.uwaterloo.ca/research/tr/2006/CS-2006-06.pdf.

[8] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Procedings of the 6th International Conference on Programming*. LNCS 167, 1984.

[9] John C. Reynolds. Towards a theory of type structure. In *Paris Colloquium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 157–168. Springer, 1974.

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

[11] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.