Type Preservation as a Confluence Problem

Aaron Stump, Garrin Kimmell, and Ruba El Haj Omar

Computer Science
The University of Iowa
astump@acm.org

Computer Science
The University of Iowa
gkimmell@cs.uiowa.edu

Computer Science
The University of Iowa
roba-elhajomar@uiowa.edu

Abstract

This paper begins with recent work by Kuan, MacQueen, and Findler, which shows how standard type systems, such as the simply typed lambda calculus, can be viewed as abstract reduction systems operating on terms. The central idea is to think of the process of typing a term as the computation of an abstract value for that term. The standard metatheoretic property of type preservation can then be seen as a confluence problem involving the concrete and abstract operational semantics, viewed as abstract reduction systems (ARSs).

In this paper, we build on the work of Kuan et al. by showing show how modern ARS theory, in particular the theory of decreasing diagrams, can be used to establish type preservation via confluence. We illustrate this idea through several examples of solving such problems using decreasing diagrams. We also consider how automated tools for analysis of term-rewriting systems can be applied in testing type preservation.

1 Introduction

The central idea of this paper is to view typing as an abstract operational semantics, and then study its interaction with concrete operational semantics using the theory of abstract reduction systems (ARSs). This idea was already proposed by Kuan, MacQueen, and Findler in 2007 [10]. They did not, however, use modern tools of term-rewriting theory in their study. Ellison et al. also explored a similar rewriting approach to type inference, but again without applying term-rewriting theory for the metatheory [7] (also [9]). In contrast, we seek to apply powerful tools from term-rewriting theory to prove standard metatheoretic properties of type systems.

Like Kuan et al., we view typing as abstract reduction, and consider types to be abstract forms of the values produced by reduction. For example, the concrete term, $\lambda x : \texttt{int}.x$ reduces to the abstract value $\texttt{int} \Rightarrow \texttt{int}$. Reduction is defined on *mixed terms*, which contain both (standard) concrete terms, and partially abstract ones. We show that the combined relation consisting of both abstract and concrete reduction is confluent, for typed terms. Kuan et al. also claim this result, but their proof sketch appeals to the standard metatheoretic property of Type Preservation. In contrast, we prove confluence directly using decreasing diagrams, and show how this then implies the standard metatheoretic property of Type Preservation.

The contribution of this paper is to show how type preservation, cast as a confluence problem, can be solved using the tools of abstract reduction systems and term-rewriting. This provides an alternative proof method for establishing type preservation for programming languages. Having alternative methods is, of course, valuable in its own right, but we will see that the rewriting approach is qualitatively simpler than the traditional one, and arguably more intuitive. This may enable shorter and simpler proofs of type preservation for other systems, as we consider further in the Conclusion.

We begin (Section 2) by developing the approach in detail for a straightforward example, namely call-by-value Simply Typed Lambda Calculus (STLC). We consider next several variations on this, including extending STLC with a fixed-point combinator (Section 3), adding polymorphism (Section 4), and implementing type inference (Section 5). Kuan et al. also considered type inference, but did not make the connection which we identify, that type inference from the rewriting perspective corresponds to narrowing, rather than rewriting. These first systems we consider can all can be analyzed using a very simple labeling for the decreasing diagrams argument. We conclude with a trickier example, namely simply typed combinators with uniform syntax (i.e., no syntactic distinction between terms and types) in Section 6. We show how automated term-rewriting tools can be used to partially automate the proof of type preservation.

2 A Rewriting View of Simple Typing

This section demonstrates the proposed new approach, for the example of STLC. While our focus in this paper is Type Preservation, we also consider the standard Progress and Type Safety properties. For Progress, it is instructive to include reduction rules for some selected constants (a and f below), so that there are stuck terms that should be ruled out by the type system. Otherwise, in pure STLC, every closed normal form is a value, namely a λ-abstraction. We see how to view a type-computation (also called type-synthesis) system for STLC as an abstract operational semantics, and type preservation as a form of confluence for the combination of the abstract and the standard concrete operational semantics. We first recapitulate the standard approach to the definitions.

We use several standard notations in this paper. For an abstract reduction relation $\to$, we define $\to^{=}$ as its reflexive closure, $\to^{+}$ as its transitive closure, and $\to^{*}$ as its reflexive-transitive closure.

2.1 Syntax and Semantics

The syntax for terms, types, and contexts is the following, where f and a are specific constants, and x ranges over a countably infinite set of variables:

$$\begin{array}{lll}
\textit{types } T & ::= & A \mid T_1 \Rightarrow T_2 \\
\textit{standard terms } t & ::= & f \mid a \mid x \mid t_1\ t_2 \mid \lambda x : T.t \\
\textit{contexts } \Gamma & ::= & \cdot \mid \Gamma, x : T
\end{array}$$

We assume standard additional conventions and notations, such as $[t/x]t'$ for the capture-avoiding substitution of t for x in t', and $E[t]$ for grafting a term into a context. Figure 1 defines a standard type system for STLC, which can be viewed as deterministically computing a type T as output, given a term t and a typing context Γ as inputs. A standard small-step call-by-value (CBV) operational semantics is defined using the rules of Figure 2.

As mentioned above, we are including constants so that the Progress theorem is not trivial. These constants are a and f, with the reduction rule $E_c[fa] \to_c E_c[a]$. Using these constants we can also construct stuck terms, such as ff, which we demonstrate are ill-typed in the proof of Progress. An example concrete reduction is (with redexes underlined):

$$\underline{(\lambda x : (A \to A).x\ (x\ a))\ f} \ \to_c\ f\ \underline{(f\ a)}\ \to_c\ \underline{f\ a}\ \to_c\ a$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{}{\Gamma \vdash f : A \Rightarrow A} \qquad \frac{}{\Gamma \vdash a : A}$$

$$\frac{\Gamma \vdash t_1 : T_2 \Rightarrow T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T_1} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1 . t : T_1 \Rightarrow T_2}$$

Figure 1 Type-computation rules for simply typed lambda calculus with selected constants

$$\frac{}{E[(\lambda x : T. t)\ v] \ \to\ E[[v/x]t]} \qquad \begin{array}{rcl} \textit{values } v & ::= & \lambda x : T.t \mid a \mid f \\ \textit{evaluation contexts } E & ::= & * \mid (E\ t) \mid (v\ E) \end{array}$$

$$\frac{}{E[f\ a] \ \to\ E[a]}$$

Figure 2 A call-by-value small-step operational semantics

2.2 Basic Metatheory

The main theorem relating the reduction relation $\to$ and typing is **Type Preservation**, which states:

$$(\Gamma \vdash t : T \ \wedge\ t \to t') \ \Rightarrow\ \Gamma \vdash t' : T$$

The standard proof method is to proceed by induction on the structure of the typing derivation, with case analysis on the reduction derivation (cf. Chapters 8 and 9 of [12]). A separate induction is required to prove a substitution lemma, needed critically for type preservation for β-reduction steps:

$$\Gamma \vdash t : T \ \wedge\ \Gamma, x : T \vdash t' : T' \ \Rightarrow\ \Gamma \vdash [t/x]t' : T'$$

One also typically proves **Progress**:

$$(\cdot \vdash t : T \ \wedge\ t \not\to) \ \Rightarrow\ \exists v.\, t = v$$

Here, the notation $t \not\to$ means $\forall t'.\ \neg(t \ \to\ t')$; i.e., t is a normal form. Normal forms which are not values are called *stuck* terms. An example is $f\ f$. Putting Type Preservation and Progress together allows us to prove **Type Safety** [18]. This property states that the normal forms of closed well-typed terms are values, not stuck terms, and in our setting can be stated this way:

$$(\cdot \vdash t : T \ \wedge\ t \to^* t' \not\to) \ \Rightarrow\ \exists v.\, t' = v$$

This is proved by induction on the length of the reduction sequence from t to t'.

2.3 Typing as Abstract Operational Semantics

To view typing as an abstract form of reduction, we use mixed terms, defined in Figure 3. Types like $T_1 \Rightarrow T_2$ will serve as abstractions of λ-abstractions. Figure 4 gives rules for concrete ($\to_c$) and abstract ($\to_a$) reduction. We denote the union of these reduction relations as $\to_{ca}$. The definition of abstract evaluation contexts makes abstract reduction nondeterministic, as reduction is allowed anywhere inside a term. This is different from the approach followed by Kuan et al., where abstract and concrete reduction are both deterministic. Here is an example reduction using the abstract

$$
\begin{array}{lll}
\textit{types } T & ::= & A \mid T_1 \Rightarrow T_2 \\
\textit{standard terms } t & ::= & x \mid \lambda x : T.\, t \mid t\; t' \mid a \mid f \\
\textit{mixed terms } m & ::= & x \mid \lambda x : T.\, m \mid m\; m' \mid a \mid f \mid \\
 & & A \mid T \Rightarrow m \\
\textit{standard values } v & ::= & \lambda x : T.t \mid a \mid f \\
\textit{mixed values } u & ::= & \lambda x : T.m \mid T \Rightarrow m \mid A \mid a \mid f
\end{array}
$$

Figure 3 Syntax for STLC using mixed terms

$$
\frac{}{E_c[f\ a] \ \rightarrow_c \ E_c[a]}\ c(f\text{-}\beta) \qquad \frac{}{E_c[(\lambda x : T.\, m)\ u] \ \rightarrow_c \ E_c[[u/x]m]}\ c(\beta)
$$

$$
\frac{}{E_a[(T \Rightarrow m)\ T] \ \rightarrow_a \ E_a[m]}\ a(\beta) \qquad \frac{}{E_a[\lambda x : T.\, m] \ \rightarrow_a \ E_a[T \Rightarrow [T/x]m]}\ a(\lambda)
$$

$$
\frac{}{E_a[f] \ \rightarrow_a \ E_a[A \Rightarrow A]}\ a(f) \qquad \frac{}{E_a[a] \ \rightarrow_a \ E_a[A]}\ a(a)
$$

$$
\begin{array}{lll}
\textit{mixed evaluation contexts } E_c & ::= & * \mid (E_c\ t) \mid (u\ E_c) \\
\textit{abstract evaluation contexts } E_a & ::= & * \mid (E_a\ m) \mid (m\ E_a) \mid \lambda x : T.\, E_a \mid T \Rightarrow E_a
\end{array}
$$

Figure 4 Abstract and concrete operational semantics for STLC

operational semantics:

$$
\begin{array}{l}
\lambda x : (A \Rightarrow A).\, \lambda y : A.\, (x\ (x\ y)) \ \rightarrow_a \\
\lambda x : (A \Rightarrow A).\, A \ \Rightarrow \ (x\ (x\ A)) \ \rightarrow_a \\
(A \Rightarrow A) \ \Rightarrow \ A \ \Rightarrow ((A \Rightarrow A)\ ((A \Rightarrow A)\ A)) \ \rightarrow_a \\
(A \Rightarrow A) \ \Rightarrow \ A \ \Rightarrow ((A \Rightarrow A)\ A) \ \rightarrow_a \\
(A \Rightarrow A) \ \Rightarrow \ A \ \Rightarrow A
\end{array}
$$

The final result is a type T. Indeed, using the standard typing rules of Section 2.1, we can prove that the starting term of this reduction has that type T, in the empty context. Abstract reduction to a type plays the role of typing above.

If we look back at our standard typing rules (Figure 1), we can now see them as essentially big-step abstract operational rules. Recall that big-step CBV operational semantics for STLC is defined by:

$$
\frac{t_1 \Downarrow \lambda x : T.t_1' \quad t_2 \Downarrow t_2' \quad [t_2'/x]t_1' \Downarrow t'}{t_1\ t_2 \Downarrow t'}
$$

In our setting, this would be concrete big-step reduction, which we might denote $\Downarrow_c$. The abstract version of this rule, where we abstract λ-abstractions by arrow-types, is

$$
\frac{t_1 \Downarrow_a T \Rightarrow T' \quad t_2 \Downarrow_a T}{t_1\ t_2 \Downarrow_a T'}
$$

If we drop the typing context from the typing rule for applications (from Figure 1), we obtain essentially the same rule.

The standard approach to proving type preservation relates a small-step concrete operational semantics with a big-step abstract operational semantics (i.e., the standard typing relation). We find it both more elegant, and arguably more informative to relate abstract and concrete small-step relations, as we will do in the next section.

▶ **Theorem 1** (Relation with Standard Typing)**.** *For standard terms t, we have $x_1 : T_1, \cdots, x_n : T_n \vdash t : T$ iff $[T_1/x_1, \cdots, T_n/x_n]t \to_a^* T$.*

See the Section A (in the Appendix) for the proof.

▶ **Theorem 2** (Termination of Abstract Reduction)**.** *The relation $\to_a$ is terminating.*

Proof. Whenever $m \to_a m'$, the following measure is strictly decreased from m to m': the number of occurrences of term constructs (listed in the definition of *terms*) which are not also type constructs (listed in the definition of *types*) and which occur in the term. Term constructs of STLC which are not also type constructs are constants, variables, λ-abstractions, and applications. **End proof.**

2.4 Type Preservation as Confluence

The relation $\to_{ca}$ is not confluent in general, as Kuan et al. note also in their setting. It is, however, confluent if restricted to *typable* terms, which are mixed terms m such that $m \to_a^* T$ for some type T. We will make use here of the standard notion of confluence of an element with respect to a binary relation $\to$: m is confluent (with respect to $\to$) iff for all m_1 and m_2 such that $m_2 \leftarrow^* m \to^* m_1$, there exists $\hat{m}$ such that $m_1 \to^* \hat{m} \leftarrow^* m_2$. In this section, we prove the following result:

▶ **Theorem 3** (Confluence of combined reduction)**.** *Every typable mixed term is confluent with respect to the reduction relation $\to_{ca}$.*

We obtain the following as an obvious corollary, noting that types T are in normal form (so joinability of m' and T becomes just reducibility of m' to T):

▶ **Theorem 4** (Type Preservation)**.** *If $m \to_a^* T$ and $m \to_c^* m'$, then $m' \to_a^* T$.*

We can phrase this result already in terms of multistep concrete reduction, while as described above, the standard approach to type preservation is stated first for single-step reduction, and then extended inductively to multistep reduction. Theorem 4 can also be phrased in terms of standard typing, using Theorem 1.

We prove Theorem 3 by a simple application of *decreasing diagrams* [16]. Recall that the main result of the theory of decreasing diagrams is that if every local peak can be completed to a *locally decreasing diagram* with respect to a fixed well-founded partial ordering on labeled steps in the diagram, then the ARS is confluent. A locally decreasing diagram has peak $s_1 \leftarrow_\alpha t \to_\beta s_2$ and valley of the form

$$s_1 \to^*_{\curlyvee\alpha} \to^{=}_{\beta} \to^*_{(\curlyvee\alpha)\cup(\curlyvee\beta)} \hat{t} \leftarrow^*_{(\curlyvee\alpha)\cup(\curlyvee\beta)} \leftarrow^{=}_{\alpha} \leftarrow^*_{\curlyvee\beta} s_2$$

where $\curlyvee\alpha$ denotes the set of labels strictly smaller in the fixed well-founded ordering than α, and if A is a set of labels, then $\to_A$ denotes the relation $\bigcup_{\alpha \in A} \to_\alpha$.

For Theorem 3, we label every step $m \to_c m'$ with c, and every step $m \to_a m'$ with a, using the label ordering $a < c$. We must prove that every local peak starting with a typable term can be completed to a locally decreasing diagram. Since $\to_c$ is deterministic (due to the restrictions inherent in the definition of reduction contexts E_c), we must only consider local peaks of the form $m_1 \leftarrow_a m \to_a m_2$ (we will call these *aa*-peaks) and $m_1 \leftarrow_a m \to_c m_2$ (*ac*-peaks). For the first, we have the following theorem:

▶ **Theorem 5.** *The relation $\to_a$ is confluent.*

Proof. In fact, we can prove that $\to_1$ has the diamond property (i.e., $(\leftarrow_a \cdot \to_a) \subseteq (\to_a \cdot \leftarrow_a)$, which is well-known to imply confluence). Suppose $m \to_a m_1$ and $m \to_a m_2$. No critical overlap is possible between these steps, because none of the redexes in the a-rules of Figure 4 (such as $(T \Rightarrow m)\ T$ in the $a(\beta)$ rule) can critically overlap another such redex. If the positions of the redexes in the terms are parallel, then (as usual) we can join m_1 and m_2 by applying to each the reduction required to obtain the other. Finally, we must consider the case of non-critical overlap (where the position of one redex in m is a prefix of the other position). We can also join m_1 and m_2 in this case by applying the reduction to m_i which was used in $m \to_a m_{3-i}$, because abstract reduction cannot duplicate or delete an a-redex. The only duplication of any subterm in the abstract reduction rules of Figure 4 is of the type T in $a(\lambda)$. The only deletion possible is of the type T in $a(\beta)$. Since types cannot contain redexes, there is no duplication or deletion of redexes. This means that if the position of the first redex is a prefix of the second (say), then there is exactly one descendant (see Section 4.2 of [14]) of the second redex in m_1, and this can be reduced in one step to join m_1 with the reduct of m_2 obtained by reducing the first redex. So every *aa*-peak can be completed with one joining step on each side of the diagram (and local decreasingness obviously holds). This gives us the diamond property and thus confluence for $\to_a$ already at this point. **End proof.**

We return now to the rest of the proof of Theorem 3, and consider the *ac*-peaks. There are only two possibilities. First, we could have the a-redex at a position parallel to the position of the c-redex. In this case, the diagram can be appropriately completed by commuting the steps. Second we could have the a-redex inside a subterm of the c-redex. There are four simple situations, and one somewhat more complex situation. The first two simple situations are that the a-redex is inside m or inside mixed value u, respectively, where the redex is $(\lambda x : T.\, m)\ u$. In the first case, the peak is

$$E_c[(\lambda x : T.\, m')\ u] \ \leftarrow_a \ E_c[(\lambda x : T.\, m)\ u] \ \to_c \ E_c[[u/x]m]$$

The required valley is just:

$$E_c[(\lambda x : T.\, m')\ u] \ \to_c \ E_c[[u/x]m'] \ \leftarrow_a \ E_c[[u/x]m]$$

The right joining step is justified because abstract reduction is closed under substitution. The labels on the single joining sides are as required for local decreasingness. In the second simple situation, the peak is:

$$E_c[(\lambda x : T.\, m)\ u'] \ \leftarrow_a \ E_c[(\lambda x : T.\, m)\ u] \ \to_c \ E_c[[u/x]m]$$

The required valley is:

$$E_c[(\lambda x : T.\, m)\ u'] \ \to_c \ E_c[[u'/x]m] \ \leftarrow_a^* \ E_c[[u/x]m]$$

Here, the labels on the right joining path are all less than the label on the right edge of the peak, satisfying the requirement for local decreasingness. Note that the left joining step would not be possible if we had phrased concrete reduction using just standard terms: we need to apply the $c(\beta)$ rule with mixed value u, which might not be a standard value. Also, we require the following easily proved lemma (proof omitted), to conclude that contracting the a-redex in u indeed results in a new value u':

▶ **Lemma 6.** *If $u \to_a^* m$ then m is also a mixed value.*

$$\begin{tikzcd}
& E_c[(\lambda x : T_1.m)\ u] \arrow[dl, "a"'] \arrow[dr, "c"] & \\
E_c[(T_1 \Rightarrow [T_1/x]m)\ u] \arrow[d, "\textit{since } u \to_a^* T_1\ \ a"', "*"] & & E_c[[u/x]m] \arrow[ddl, "a"', "*", "\textit{since } u \to_a^* T_1"] \\
E_c[(T_1 \Rightarrow [T_1/x]m)\ T_1] \arrow[dr, "a"'] & & \\
& E_c[[T_1/x]m] &
\end{tikzcd}$$

Figure 5 Crucial locally decreasing diagram for STLC

The second two simple situations involve $f\ a$. First:

$$E_a[(A \Rightarrow A)\ a] \leftarrow_a E_a[f\ a] \rightarrow_c E_a[a]$$

The joining valley, which is again locally decreasing because its labels are less than c, is:

$$E_a[(A \Rightarrow A)\ a] \rightarrow_a E_a[(A \Rightarrow A)\ A] \rightarrow_a E_a[A] \leftarrow_a E_a[a]$$

Second:

$$E_a[f\ A] \leftarrow_a E_a[f\ a] \rightarrow_c E_a[a]$$

The joining valley is:

$$E_a[f\ A] \rightarrow_a E_a[(A \Rightarrow A)\ A] \rightarrow_a E_a[A] \leftarrow_a E_a[a]$$

Finally, the more complicated case is shown in Figure 5. Since the term at the peak is typable, we know that $u \rightarrow_a^* T_1$. This is because abstract reduction cannot eliminate an application from a term, except via $a(\beta)$: no abstract reduction rule can erase a term. So we know that $u \rightarrow_a^* T_1$, or else the application displayed at the peak could not be eliminated by abstract reduction.

Considering the diagram in Figure 5, we see that again, the labels in the joining steps are all less than the label on the right edge of the peak. So the condition for local decreasingness is satisfied. Note that the bottom-right joining step, from $E_c[[u/x]m]$ to $E_c[[T_1/x]m]$ requires nondeterminism of $\rightarrow_a$ (and if $x \notin \mathit{Vars}(m)$, is just an identity step). It would not hold if abstract reduction were restricted to the abstract analog of call-by-value concrete reduction, as done by Kuan et al. We have proved the following, which suffices by the theory of decreasing diagrams to prove **Confluence** (Theorem 3):

▶ **Lemma 7** (Local Decrease for Type Diagrams)**.** *Every local peak of $\rightarrow_{ac}$ can be completed to a locally decreasing diagram.*

We can complete the basic metatheory for STLC by proving:

▶ **Theorem 8** (Progress)**.** *If standard term t is closed, $t \rightarrow_a^* T$, and $t \not\rightarrow_c$, then t is a concrete standard value.*

Proof. The only possibility for a closed standard term that is a normal form of $\to_c$ other than a standard value is a stuck term $E_c[d\ t]$, where t is a c-normal form and either $d \equiv a$, or else $d \equiv f$ and $t \not\equiv a$. Let $d\ t$ be the smallest such stuck term in $E_c[d\ t]$. The term $E_c[d\ t]$ has an a-normal form (by Theorem 2), which must contain a descendant of $d\,t$. This is because, as already noted, our abstract reduction rules drop an expression only if it is a type (rule $a(\beta)$). Also, abstract reduction cannot contract a descendant of $f\ t$ itself, which we argue by cases on the form of $f\ t$ (call this term $\hat{t}$). If $\hat{t} \equiv a\ t$, then for some m, $\hat{t}$ a-normalizes to $A\ m$, which is still a descendant of $\hat{t}$. If $\hat{t} \equiv f\ f$, then $\hat{t}$ a-normalizes to $(A \Rightarrow A)\ (A \Rightarrow A)$ (also still a descendant). If $\hat{t} \equiv f\ \lambda x : T.t$, then $\hat{t}$ a-normalizes to $(A \Rightarrow A)\ (A \Rightarrow m)$ for some m. This shows that a descendant of $d\,t$ must still be contained in the a-normal form of $t \equiv E_c[d\ t]$, contradicting $t \to_a^* T$. **End proof.**

We then obtain the final result as a direct corollary of Theorems 4 and 8.

▶ **Theorem 9** (Type Safety). *If standard term t is closed, $t \to_a^* T$, and $t \to_c^* t' \not\to_c$, then t' is a concrete value.*

The development in this section compares favorably with the standard one, which requires an induction proof for type preservation, another for the additionally required substitution lemma, and a final one for Type Safety. In contrast, here all that was required was to analyze ac-peaks for local decreasingness (the aa-peaks being easily joinable in one step) in order to establish Type Preservation. That analysis is both more local and more informative, as it gives a more direct insight into how the (small-step) process of abstracting a term to its type relates to the small-step operational semantics.

3 STLC with a Fixed-Point Combinator

We may easily extend the results of the previous section to include a fixed-point combinator *fix*. This is a standard example, which enables now possibly diverging concrete reductions. The approach using decreasing diagrams easily adapts to this new situation. Figure 6 shows the additions to the syntax of STLC. The proof of **Termination of Abstract Reduction** (Theorem 2) goes through exactly as stated above, since *fix* is a term construct but not a type construct, and our new abstract reduction for *fix*-terms again reduces the number of occurrences of term constructs which are not type constructs. For **Local Decrease for Type Diagrams** (Lemma 7), there is still no critical overlap between a-steps, and no possibility of erasing or duplicating a redex, so $\to_a$ still has the diamond property. The simple ac-peaks are easily completed, similarly to the simple ones for STLC. We must then just consider this new ac-peak:

$$E_c[(T \Rightarrow T)\ [T/f]m) \leftarrow_a \underline{E_c[\mathit{fix}\ f : T.\, m]} \to_a E_c[[\mathit{fix}\ f : T.\, m/f]m]$$

This can be completed as follows:

$$\begin{array}{ll} L. & E_c[(T \Rightarrow T)\ [T/f]m] \to_a^* E_c[(T \Rightarrow T)\ T] \to_a E_c[T] \\ R. & E_c[[\mathit{fix}\ f : T.\, m/f]m] \to_a^* E_c[[((T \Rightarrow T)\ [T/f]m)/f]m] \to_a^* E_c[[T/f]m] \to_a^* E_c[T] \end{array}$$

Here, several steps use $[T/f]m \to_a^* T$, which holds because the term at the peak is typable. This diagram is again locally decreasing. We conclude Confluence (Theorem 3) as above. There are no possible additional stuck terms, so **Progress** (Theorem 8) is trivially extended, allowing us to conclude **Type Safety** (Theorem 9).

4 Adding Polymorphism

In this section, we extend STLC with System-F style polymorphism. Figures 7 and 8 show the additional syntax, evaluation contexts, and reduction rules. At the term level, we add syntax for type

$$\begin{array}{lcl}
\textit{(standard) terms } t & ::= & \ldots \mid \textit{fix } f : T.\, t \\
\textit{(mixed) terms } m & ::= & \ldots \mid \textit{fix } f : T.\, m \\
\textit{abstract evaluation contexts } E_a & ::= & \ldots \mid \textit{fix } f : T.\, E_a
\end{array}$$

$$\frac{}{E_c[\textit{fix } f : T.\, m] \;\to_c\; E_c[[\textit{fix } f : T.\, m/f]m]} \qquad \frac{}{E_a[\textit{fix } f : T.\, m] \;\to_a\; E_a[(T \Rightarrow T)\ [T/f]m]}$$

Figure 6 Extending STLC with a fixed-point combinator

$$\begin{array}{lcl}
\textit{types } T & ::= & \ldots \mid \alpha \mid \forall\alpha.T \\
\textit{standard terms } t & ::= & \ldots \mid \Lambda\alpha.t \mid t@T \\
\textit{mixed terms } m\ t & ::= & \ldots \mid \Lambda\alpha.m \mid m@T \mid \forall\alpha.m \\
\textit{standard values } v & ::= & \ldots \mid \Lambda\alpha.t \\
\textit{mixed values } u & ::= & \ldots \mid \Lambda\alpha.m \mid \forall\alpha.m
\end{array}$$

Figure 7 Polymorphism syntactic extensions

abstraction (Λ), type application $(t@t)$, and type variables α. At the type level, we add universally quantified types $\forall\alpha.\,T$. As is standard practice, concrete reduction now includes a β-like rule for eliminating type applications of type abstractions. The abstract reduction relation has a similar rule for type applications of universal types, and also a rule for computing a universal type from a type abstraction.

The argument for **Termination of Abstraction Reduction** (Theorem 2) is extended easily, since the two new $\to_a$ rules, $a(\Lambda)$ and $a(\beta_T)$, again strictly decrease the number of occurrences of terms which are not types. We can extend the proof of **Local Decrease for Type Diagrams** (Lemma 7) by considering new $\to_{ac}$ peaks. As before, these can be separated into cc, aa, and ac cases. As with the STLC case, the $\to_c$ relation is deterministic, so there are no cc peaks. There are no critical overlaps between $\to_a$ steps, and again no possibility to erase or duplicate a redex. So we maintain the diamond property for aa-peaks.

A c-redex of the form $(\Lambda x.\,t)@T$ gives rise to a possible ac-peak, where a-reduction occurs under the Λ:

$$E_c[(\Lambda\alpha.\,m')@T] \;\leftarrow_a\; E_c[(\Lambda\alpha.\,m)@T] \to_c\; E_c[[T/\alpha]m]$$

$$\frac{}{E_c[(\Lambda\alpha.m)@T] \;\to_c\; E_c[[T/\alpha]m]}\ c(\beta_T) \qquad \frac{}{E_a[(\forall\alpha.m)@T] \;\to_a\; E_a[[T/\alpha]m]}\ a(\beta_T)$$

$$\frac{}{E_a[\Lambda\alpha.t] \;\to_a\; E_a[\forall\alpha.t]}\ a(\Lambda)$$

$$\begin{array}{lcl}
\textit{mixed evaluation contexts } E_c & ::= & \ldots \mid (E_c\ @\ T) \mid (u\ @\ E_c) \\
\textit{abstract evaluation contexts } E_a & ::= & \ldots \mid (E_a\ m) \mid ((\forall\alpha.m)@\ E_a)
\end{array}$$

Figure 8 Polymorphism operational semantics extensions

As with the STLC, a-reduction is closed under substitution, so this peak can be completed as shown:

$$E_c[(\Lambda\alpha.\, m')@T] \;\to_c\; E_c[[T/\alpha]m'] \leftarrow_a\; E_c[[T/\alpha]m]$$

The grammar for the polymorphic language requires that all type-level applications contain a proper type T in the argument position, so there does not exist a ac-peak analagous to the STLC ac-peak where an a-step occurs in the argument subterm of a c-redex. There is only one critical overlap, due to the $a(\Lambda)$ and $c(\beta_T)$ rules,

$$E_c[(\forall\alpha.m)@T] \leftarrow_a\; E_c[(\Lambda\alpha.m)@T] \to_c E_c[[T/\alpha]m]$$

This can be completed with a single c-reduction:

$$E_c[(\forall\alpha.m)@T] \to_c E_c[[T/\alpha]m] \leftarrow_c^= E_c[[T/\alpha]m]$$

This completes the proof for **Local Decrease of Type Diagrams**.

Finally, the proof of **Progress** for STLC is extended by considering a few new stuck terms. These are terms of the form $a@T$ and $f@T$, which reduce to non-type a-normal forms $A@T$ and $(A \Rightarrow A)@T$, respectively, contradicting Progress's assumption of typability. Also, we could have $a\ \Lambda\alpha.m$ or $f\ \Lambda\alpha.m$, but these reduce respectively to non-type a-normal forms $A\ \forall\alpha.m'$ and $(A \Rightarrow A)\ \forall\alpha.m'$, for some m'. We again conclude **Type Safety**.

5 Type Inference for STLC

We can modify STLC reduction rules to allow us to infer, rather than check, the type of STLC terms where type annotations are omitted for λ-bound variables. Inferring simple types is a central operation in ML-style type inference. The central idea is to base abstract reduction on narrowing, rather than just rewriting. Our focus in this paper is the use of confluence to prove type safety, and not necessarily the efficient implementations of the resulting reduction system. Nevertheless, we believe that abstract reduction systems can serve as the basis for efficient implementation, as demonstrated by Kuan [11] for similar reduction systems.

Figure 9 shows the grammar of STLC-inf. In this language, λ-bound variables do not have type annotations, as they will be inferred. The syntax of types now includes type variables α_i, which may be instantiated by narrowing.

Using narrowing, we can define the abstract reduction system shown in Figure 9 to infer the types of terms. In the $a(\beta)$ rule, we calculate the most general unifier of the function type's domain and the argument, and apply it to the entire term, including the evaluation context. This contrasts with the rules we have seen in previous systems presented in this paper, where substitutions are applied only to the focus of the evaluation context. We assume for all rules that introduce new type variables that they do so following a fixed order. We include the rule $a(\textit{rename})$ to avoid non-confluence due to different choices of new type variables in the rules $a(\textit{gen})$ and $a(\lambda)$.

The following theorem relates STLC-inf typing to STLC typing. The *erasure* of an STLC term, $|t|$, drops type annotations from λ bindings, producing an STLC-inf term.

▶ **Theorem 10** (Relation with STLC Typing). *If $t \to_a^* T$, where T contains free type variables $\alpha_1, ..., \alpha_n$, then for all types $T_1, ...T_n$, there exists a term t' in STLC such that $t' \to_a^* [T_1/\alpha_1, ...T_n/\alpha_n]T$ in STLC and $|t'| = t$.*

See Section B (in the Appendix) for the proof.

We extend the metatheoretic results to STLC-inf as follows. We no longer have **Termination of Abstract Reduction**, due to $a(\textit{rename})$. This does not impact subsequent results, as aa-peaks are

$$\begin{array}{lll}
\textit{types } T & ::= & T \Rightarrow T \mid \alpha_i \mid A \\
\textit{standard terms } t & ::= & x \mid t\,t' \mid \lambda x.t \mid a \mid f \\
\textit{mixed terms } m & ::= & x \mid m\,m' \mid \lambda x.m \mid a \mid f \mid T \Rightarrow m \mid A \mid \alpha_i \\
\textit{standard values } v & ::= & a \mid f \mid \lambda x.t \\
\textit{mixed values } u & ::= & a \mid f \mid \lambda x.m \mid T \Rightarrow u \mid A \\
\textit{mixed evaluation contexts } E_c & ::= & * \mid (E_c\ m) \mid (u\ E_c) \\
\textit{abstract evaluation contexts } E_a & ::= & * \mid (E_a\ m) \mid (m\ E_a) \mid T \Rightarrow E_a \mid \alpha_i\ E_a
\end{array}$$

$$\frac{}{E_c[(\lambda x.m)u] \ \rightarrow_c \ E_c[[u/x]m]}\ c(\beta) \qquad \frac{\alpha_i \notin FV(E_a[\lambda x.m])}{E_a[\lambda x.m] \ \rightarrow_a \ E_a[\alpha_i \Rightarrow [\alpha_i/x]m]}\ a(\lambda)$$

$$\frac{\sigma \textit{ is } mgu(T_1, T_2)}{E_a[(T_1 \Rightarrow m)T_2] \ \rightarrow_a \ \sigma(E_a[m])}\ a(\beta) \qquad \frac{}{E_a[f] \rightarrow_a E_a[A \Rightarrow A]}\ a(f)$$

$$\frac{\alpha_j \notin FV(E_a[\alpha_i\ T])}{E_a[\alpha_i\ T] \rightarrow_a [(T \Rightarrow \alpha_j)/\alpha_i](E_a[\alpha_j])}\ a(gen) \qquad \frac{}{E_a[a] \rightarrow_a E_a[A]}\ a(a)$$

$$\frac{\rho \textit{ is a permutation of type variables}}{t \rightarrow_a \rho\ t}\ a(\textit{rename})$$

Figure 9 Type Inference Syntax and Semantics

still joinable in one step. The ac-peak depicted in Figure 5 is adapted to account for the inference of types for λ-bound variables, as shown in Figure 10. The completion of the a reduction relies on the assumption that the generated type variable α is not free in the original term.

Because the $a(\beta)$ rule uses narrowing to compute λ-bound variable types, it is possible to generate aa-peaks when a polymorphic function is used at multiple monotypes. Consider the following reduction resulting in a mixed term:

$$\begin{array}{l}(\lambda g.\lambda x.\lambda y.\lambda h.h\ (g\ x)\ (g\ y))\ (\lambda w.w)\ f\ a \quad \rightarrow_a^+ \\ \alpha_h(\alpha_g(A \Rightarrow A))(\alpha_g A)\end{array}$$

We can reduce this latter term to the following distinct stuck terms, depending on which application of α_g we contract with the *a(gen)* rule first:

$$\begin{array}{l}\alpha_h\ ((A \Rightarrow \alpha_j)\ (A \Rightarrow A))\ \alpha_j \\ \alpha_h\ \alpha_j\ (((A \Rightarrow A)\ \Rightarrow \alpha_j)\ A)\end{array}$$

But we are proving **Confluence** only for typeable terms $t \rightarrow_a^* T$. So such peaks can be disregarded. We conclude **Confluence** and then **Type Preservation** as above. The proof of **Progress** is adapted directly from STLC, as the changes in STLC-inf do not essentially affect the form of stuck terms. So we can conclude **Type Safety**.

6 Simply Typed Combinators with Uniform Syntax

In this section, we consider a language, which we call Uniform-STC, that does not make a syntactic distinction between terms and types. More advanced type systems, such as Pure Type Systems, often lack such a distinction, and rely on the typing rules to determine which expressions from the

$$\begin{tikzcd}
 & E_c[(\lambda x.m)\ u] \arrow[dl, "a"'] \arrow[dr, "c"] & \\
E_c[(\alpha \Rightarrow [\alpha/x]m)\ u] \arrow[d, "\text{since } u \to_a T"', "a"' pos=0.8] & & E_c[[u/x]m] \arrow[ddd, "\text{since } u \to_a T", "a" pos=0.8] \\
E_c[(\alpha \Rightarrow [\alpha/x]m)\ T] \arrow[d, "a"'] & & \\
{[T/\alpha]E_c[[\alpha/x]m]} \arrow[drr, "\text{since } \alpha \notin FV(E_c[(\lambda x.m)u])"', "a"' pos=0.8] & & \\
 & & E_c[[T/x]m]
\end{tikzcd}$$

Figure 10 Crucial locally decreasing diagram for STLC-inf

$$\begin{array}{lll}
\textit{mixed terms } t & ::= & S\langle t_1, t_2, t_3\rangle \mid K\langle t_1, t_2\rangle \mid t_1\ t_2 \mid t_1\ \Rightarrow\ t_2 \mid A \mid \textit{kind}(t_1,\ t_2) \\
\textit{mixed values } u & ::= & S\langle t_1, t_2, t_3\rangle \mid K\langle t_1, t_2\rangle \mid A \mid t_1 \Rightarrow t_2 \\
\textit{concrete evaluation contexts } E_c & ::= & * \mid E_c\ t \mid u\ E_c
\end{array}$$

Figure 11 Uniform-STLC language syntax and evaluation contexts

uniform syntax are types (or kinds or superkinds, etc.), and which are terms (cf. [4]). Uniform-STC is a simply typed language with uniform syntax, which we study to explore the issues that may arise in applying the present approach to more advanced type systems. The technical issue is that now we must implement kinding (i.e., type checking of types) as part the abstract reduction relation. We adopt a combinatory formulation so that the abstract reduction relation can be described by a first-order term-rewriting system.

Figure 11 shows the syntax for the Uniform-STC language. There is a single syntactic category t for mixed terms and types, which include a base type A and simple function types. $S\langle t_1, t_2, t_3\rangle$ and $K\langle t_1, t_2\rangle$ are the usual combinators, indexed by terms which determine their simple types. The *kind* construct for terms is used to implement kinding. The rules for concrete and abstract reduction are given in Figure 12. The concrete rules are just the standard ones for call-by-value reduction of combinator terms. For abstraction reduction, we are using first-order term-rewriting rules (unlike for previous systems). This has the advantage that we can apply automated analysis tools for first-order rewriting, to the abstract reduction rules. We could have also done this for the systems above, but abstract reduction there was so easily analyzed there was no need. Here, the situation is a little more complicated.

For STLC (Section 2), abstract β-redexes have the form $(T \Rightarrow t)\ T$. For Uniform-STC, since there is no syntactic distinction between terms and types, abstract β-redexes take the form $(t_1 \Rightarrow t_2)\ t_1$, and we must use kinding to ensure that t_1 is a type. This is why the $a(\beta)$ rule introduces a *kind*-term. We also enforce kinding when abstracting simply typed combinators $S\langle t_1, t_2, t_3\rangle$ and

$$\begin{array}{ll}
c(\beta\text{-}S). & \overline{E_c[S\langle t_1, t_2, t_3\rangle\ u\ u'\ u''] \to_c E_c[u\ u''\ (u'\ u'')]} \\[1em]
c(\beta\text{-}K). & \overline{E_c[K\langle t_1, t_2\rangle\ u\ u'] \to_c E_c[u]} \\[1em]
a(S). & S\langle t_1, t_2, t_3\rangle \to_a \mathit{kind}(t_1, \mathit{kind}(t_2, \mathit{kind}(t_3, (t_1 \Rightarrow t_2 \Rightarrow t_3) \Rightarrow (t_1 \Rightarrow t_2) \Rightarrow (t_1 \Rightarrow t_3)))) \\
a(K). & K\langle t_1, t_2\rangle \to_a \mathit{kind}(t_1, \mathit{kind}(t_2, (t_1 \Rightarrow t_2 \Rightarrow t_1))) \\
a(\beta). & (t_1 \Rightarrow t_2)\ t_1 \to_a \mathit{kind}(t_1, t_2) \\
a(k\text{-} \Rightarrow). & \mathit{kind}((t_1 \Rightarrow t_2), t) \to_a \mathit{kind}(t_1, \mathit{kind}(t_2, t)) \\
a(k\text{-}A). & \mathit{kind}(A, t) \to_a t
\end{array}$$

Figure 12 Concrete and abstract reduction rules

$K\langle t_1, t_2\rangle$ to their types. The rules for *kind*-terms ($a(k\text{-} \Rightarrow)$ and $a(k\text{-}A)$) make sure that the first term is a type, and then reduce to the second term.

Following our general procedure, we wish to show that every local peak at a typable term can be completed to a locally decreasing diagram. Here, we define typability by value u to mean abstract reduction to u where u is *kindable*, which we define as $\mathit{kind}(u, A) \to_a^* A$. This definition avoids the need to define types syntactically.

Unlike for previous systems like STLC, aa-peaks for Uniform-STC are slightly harder to analyze, due to the non-left-linear rule $a(\beta)$, where there could indeed be redexes in the expressions matching the repeated variable t_1. Fortunately, abstract reduction can be automatically analyzed for termination: APROVE reports that it can be shown terminating using a recursive path ordering [8]. This means that we can use van Oostrom's source-labeling heuristic for a-steps [17]. We label steps as follows:

$$\begin{array}{lcl}
\mathit{label}(m \to_c m') & = & c \\
\mathit{label}(m \to_a m') & = & (a, m)
\end{array}$$

Steps are then ordered by the following relation:

$$\begin{array}{l}
\forall m. c \succ (m, a) \\
\forall m, m'. m \to_a^+ m' \Leftrightarrow (m, a) \succ (m', a)
\end{array}$$

The abstract reduction rules are non-overlapping. The aa-peaks which occur can all be joined using either one a-step on either side as for STLC, or else using additional balancing steps if one of the rules applied is $a(\beta)$. In the latter case, the diagram is still decreasing due to the termination of abstract reduction.

But we can actually use even a simpler argument for aa-peaks. The automated confluence prover ACP reports that the abstract reduction relation is confluent [2]. By completeness of decreasing diagrams for countable relations (recalled in [17] from van Oostrom's PhD thesis), this means that there must exists a labeling of abstract reduction steps that allows all aa-peaks to be completed to locally decreasing diagrams. We can then extend this labeling to include c (labeling c-steps), and then simply make c bigger in the extended ordering than all the labels of a-steps.

With this ordering (or the source-labeling), we then have the following locally decreasing diagram (the one for $c(\beta\text{-}S)$ is similar and omitted), where $\hat{t}$ is $(t_1 \Rightarrow t_2 \Rightarrow t_1)$ and u is $\mathit{kind}(t_1, \mathit{kind}(t_2, *))$:

$P.$ $E_c[u[(\hat{t}\; t\; t')]] \leftarrow_a E_c[(K\langle t_1, t_2\rangle\; t\; t')] \rightarrow_a E_c[t]$
$L1.$ $E_c[u[(\hat{t}\; t\; t')]] \rightarrow_a^*$
$L2.$ $E_c[u[(\hat{t}\; t_1\; t'')]] \rightarrow_a$
$L3.$ $E_c[u[((t_2 \Rightarrow t_1)\; t'')]] \rightarrow_a^*$
$L4.$ $E_c[u[((t_2 \Rightarrow t_1)\; t_2)]] \rightarrow_a$
$L5.$ $E_c[\mathit{kind}(t_1, \mathit{kind}(t_2, t_1))] \rightarrow_a^*$
$E_c[t_1]$
$R1.$ $E_c[t] \rightarrow_a^*$
$R2.$ $E_c[t_1]$

The $\rightarrow_a^*$-steps are justified because the peak term (shown on line (P)) is assumed to be typable. By confluence of abstract reduction, this implies that the sources of all the left steps are also typable. For each $\rightarrow_a^*$-step, since abstract reduction cannot drop redexes (as all rules are non-erasing), we argue as for STLC that a descendant of the appropriate displayed *kind*-term or application must eventually be contracted, as otherwise, a stuck descendant of such would remain in the final term. Kindable terms cannot contain stuck applications or stuck *kind*-terms, because our abstract reduction rules are non-erasing. And contraction of those displayed *kind*-terms or applications requires the reductions used for the $\rightarrow_a^*$-steps, which are sufficient to complete the diagram. The diagram is again locally decreasing because the c-step from the peak is greater than all the other steps in the diagram. We thus have **Confluence** of ac-reduction for typable terms, and the following statement of type preservation (relying on our definition above of typability):

▶ **Theorem 11** (Type Preservation). *If t is typable with t_1 and $t \rightarrow_c t'$, then t' is also typable with t_1.*

As an aside, note that a natural modification of this problem is out of the range of ACP, version 0.20. Suppose we are trying to group kind-checking terms so that we can avoid duplicate kind checks for the same term. For this, we may wish to permute *kind*-terms, and pull them out of other term constructs. The following rules implement this idea, and can be neither proved confluent nor disproved by ACP, version 0.20. Just the first seven rules are also unsolvable by ACP.

```
(VAR a b c A B C D)
(RULES
  S(A,B,C) -> kind(A,kind(B,kind(C,
                arrow(arrow(arrow(A,arrow(B,C)),arrow(A,B)),arrow(A,C)))))
  K(A,B) -> kind(A,kind(B,arrow(A,arrow(B,A))))
  app(arrow(A,b),A) -> kind(A,b)
  kind(base,a) -> a
  kind(arrow(A,B),a) -> kind(A, kind(B, a))
  kind(A,kind(A,a)) -> kind(A,a)
  kind(A,kind(B,a)) -> kind(B,kind(A,a))
  app(kind(A,b),c) -> kind(A,app(b,c))
  app(c,kind(A,b)) -> kind(A,app(c,b))
  arrow(kind(A,b),c) -> kind(A,arrow(b,c))
  arrow(c,kind(A,b)) -> kind(A,arrow(c,b))
  kind(kind(a,b),c) -> kind(a,kind(b,c))
)
```

7 Conclusion

We have seen how to use decreasing diagrams to establish confluence of a reduction relation combining concrete reduction (the standard operational semantics) with abstract reduction, which can be thought of as a small-step typing relation. Type Preservation is then an immediate corollary of confluence. We have applied this to several example type systems, illustrating a variety of different features. We highlight that we are able to cast type inference as a form of narrowing, and that for systems that easily fit the first-order term rewriting format, such as combinatory systems, we can even apply standard analysis tools like termination and confluence checkers to automate part of the proof of type preservation.

There are many directions for future work in this area. First, the approach should be applied to more advanced type systems. Dependent type systems pose a particular challenge, because from the point of view of abstract reduction, Π-bound variables must play a dual role. When computing a dependent function type $\Pi x : T.\, T'$ from an abstraction $\lambda x : T.t$, we may need to abstract x to T, as for STLC; but we may also need to leave it unabstracted, since with dependent types, x is allowed to appear in the range type T'. We conjecture that this can be accommodated by substituting a pair (x, T) for x in the body of the λ-abstraction, and then choosing either the term or type part of the pair depending on how the pair is used. For type systems that use conversion relations to change the type of a term by rewriting, the current approach should be very natural. Systems like the Calculus of Constructions and its descendants rely heavily on conversion to simplify type-level redexes.

Naturally, it would be interesting to try to extend the range of automated confluence tools for use in automating type preservation proofs completely. We restricted our attention to deterministic concrete reduction relations for simplicity, which kept us out of the first-order term-rewriting format. We could certainly lift that restriction in some cases, and go to combinatory systems, to try applying first-order analysis tools. The challenge is that the reduction relations we are considering are not confluent in general, but only for the typable terms. To test this restricted form of confluence, confluence-proving algorithms would be needed where one can limit the peaks considered to those that rewrite to a certain fixed value (as this could then be used to restrict to the typable terms). While the Programming Languages community has invested substantial effort in recent years on computer-checked proofs of properties like Type Safety for programming languages (initiated particularly by the POPLmark Challenge [3]), there is relatively little work on fully automatic proofs of type preservation (an example is [13]). The rewriting approach could contribute to filling that gap.

Finally, our longer term goal is to use this approach to design and analyze type systems for more powerful forms of program analysis. In program verification tools like PEX and KEY, symbolic simulation is a central component [5, 15]. But these systems do not seek to prove that their symbolic-simulation algorithms are correct. Indeed, the authors of the KEY system argue that the effort required to verify soundness of the rules underlying symbolic simulation is better spent on other tasks [6]. This certainly suggests that the authors believe such verification would be onerous. The rewriting approach promises to make it easier to relate symbolic simulation, viewed as an abstract reduction relation, with the small-step operational semantics.

Acknowledgments. We thank the anonymous RTA 2011 reviewers for their helpful comments, which have improved this paper. We also acknowledge funding support for this work from the U.S. National Science Foundation, contract CCF-0910510, as part of the Trellys project.

References

1 S. Abramsky, D. Gabbay, and T. Maibaum, editors. *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

2 T. Aoto, J. Yoshida, and Y. Toyama. Proving Confluence of Term Rewriting Systems Automatically. In R. Treinen, editor, *Rewriting Techniques and Applications (RTA)*, pages 93–102, 2009.

3 B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark Challenge.

In *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, 2005.

4 H. Barendregt. *Lambda Calculi with Types*, pages 117–309. Volume 2 of Abramsky et al. [1], 1992.

5 B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

6 B. Beckert and V. Klebanov. Must Program Verification Systems and Calculi be Verified? In *Proceedings, 3rd International Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC), Seattle, USA*, pages 34–41, 2006.

7 C. Ellison, T. Şerbănuţă, and G. Roşu. A Rewriting Logic Approach to Type Inference. In A. Corradini and U. Montanari, editors, *Recent Trends in Algebraic Development Techniques (WADT)*, pages 135–151, 2008.

8 J. Giesl, P. Schneider-Kamp, and R. Thiemann. Automatic Termination Proofs in the Dependency Pair Framework. In U. Furbach and N. Shankar, editors, *Automated Reasoning, Third International Joint Conference (IJCAR)*, pages 281–286, 2006.

9 M. Hills and G. Rosu. A Rewriting Logic Semantics Approach to Modular Program Analysis. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11-13, 2010, Edinburgh, Scotland, UK*, pages 151–160, 2010.

10 G. Kuan, D. MacQueen, and R. Findler. A rewriting semantics for type inference. In *Proceedings of the 16th European conference on Programming (ESOP)*, pages 426–440. Springer-Verlag, 2007.

11 George Kuan. Type checking and inference via reductions. In Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt, editors, *Semantics Engineering with PLT Redex*. MIT Press, 2009.

12 B. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

13 C. Schürmann and F. Pfenning. Automated Theorem Proving in a Simple Meta-Logic for LF. In C. Kirchner and H. Kirchner, editors, *15th International Conference on Automated Deduction (CADE)*, pages 286–300, 1998.

14 TeReSe, editor. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

15 N. Tillmann and W. Schulte. Parameterized Unit Tests. *SIGSOFT Softw. Eng. Notes*, 30:253–262, 2005.

16 V. van Oostrom. Confluence by decreasing diagrams. *Theoretical Computer Science*, 126(2):259–280, 1994.

17 V. van Oostrom. Confluence by Decreasing Diagrams, Converted. In A. Voronkov, editor, *Rewriting Techniques and Applications*, pages 306–320, 2008.

18 A. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.

A Proof of Theorem 1 (Section 2.3)

Proof (⇒). We will first prove the left-to-right implication. So suppose $x_1 : T_1, \cdots, x_n : T_n \vdash t : T$. We will now prove $[T_1/x_1, \cdots, T_n/x_n]t \rightarrow_a^* T$ by induction on the structure of the typing derivation of t. To simplify the writing of the proof, we will use the following notation:

$$\begin{array}{lcl} \Gamma & = & x_1 : T_1, \cdots, x_n : T_n \\ \Gamma_{sub} & = & [T_1/x_1, \cdots, T_n/x_n] \end{array}$$

Base Case:

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

There must be some $i \in \{1, \ldots, n\}$ such that $x = x_i$ and $T = T_i$. So $\Gamma_{sub}\ x = T_i \rightarrow_a^* T_i$ as required.

Base Case:

$$\frac{}{\Gamma \vdash f : A \Rightarrow A}$$

We indeed have $f \rightarrow_a (A \Rightarrow A)$, as required. The case for $a : A$ is similar.

Case:

$$\frac{\Gamma \vdash t_1 : T_2 \Rightarrow T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T_1}$$

By the induction hypotheses for the derivations given for the two premises of this rule, we have:

$$\Gamma_{sub}\ t_1 \rightarrow_a^* T_2 \Rightarrow T_1$$
$$\Gamma_{sub}\ t_2 \rightarrow_a^* T_2$$

Our goal now is to construct the reduction sequence:

$$\Gamma_{sub}\ (t_1\ t_2) \rightarrow_a^* (T_2 \Rightarrow T_1)\Gamma_{sub}\ t_2 \rightarrow_a^* (T_2 \Rightarrow T_1)T_2 \rightarrow_a T_1$$

To construct this sequence, it is sufficient to apply transitivity of $\rightarrow_a^*$ and the following two lemmas:

▶ **Lemma 12.** *If $t_1 \rightarrow_a^* T$, then $t_1\ t_2 \rightarrow_a^* T\ t_2$ using the same number of steps.*

▶ **Lemma 13.** *If $t_2 \rightarrow_a^* T$, then $t_1\ t_2 \rightarrow_a^* t_1\ T$ using the same number of steps.*

Case:

$$\frac{\Gamma, x : T \vdash t : T'}{\Gamma \vdash \lambda x : T.\, t : T \Rightarrow T'}$$

By the induction hypothesis on the premise of this rule, we have:

$$\Gamma_{sub}\ [T/x]\ t \rightarrow_a^* T'$$

Now we need to show that

$$\Gamma_{sub}\ (\lambda x : T.\, t) \rightarrow_a^* (T \Rightarrow T')$$

We will need the following lemma to construct the reduction sequence below:

▶ **Lemma 14.** *If $t \rightarrow_a^* T'$, then $T \Rightarrow t \rightarrow_a^* T \Rightarrow T'$ using the same number of steps.*

Now, by applying one abstract(λ) and Lemma 14 we get:

$$\Gamma_{sub}\ (\lambda x : T.\, t) \rightarrow_a (T \Rightarrow \Gamma_{sub}\ [T/x]\ t) \rightarrow_a^* (T \Rightarrow T')$$

This requires the fact that $\Gamma_{sub}\ [T/x] = [T/x]\Gamma_{sub}$, which holds because $x \notin dom(\Gamma_{sub})$ since we may rename x to avoid this, and because T contains no term variables and hence is unaffected by applying Γ_{sub}.
End Proof ($\Rightarrow$).

Proof($\Leftarrow$). We will now assume $[T_1/x_1, \cdots, T_n/x_n]t \rightarrow_a^* T$ and prove $x_1 : T_1, \cdots, x_n : T_n \vdash t : T$ by induction on the number of single steps in $\rightarrow_a^*$.

Base Case: There are no single steps. This means that our term t cannot be reduced

$$\Gamma_{sub}\; t = T$$

In this case, t must be a variable (or else substitution could not result in a type T). So, $t = x$ for some variable x, where $\Gamma(x) = T$. Then we get:

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

Step Case: There is at least one single step. We proceed by case splitting on the form of t.

Case:

$$\Gamma_{sub}\; x$$

This case cannot occur, since either $x \notin dom(\Gamma_{sub})$, in which case we cannot have $x \rightarrow_a^* T$ for any type T; or else $x \in dom(\Gamma_{sub})$, and then $\Gamma_{sub}\; x = T$. We cannot have a single step in that case.

Case:

$$\Gamma_{sub}\; f$$

The only possible step is $f \rightarrow_a A \Rightarrow A$, and we indeed have $\Gamma \vdash f : A \Rightarrow A$. The case for $\Gamma_{sub}\; a$ is similar.

Case:

$$\Gamma_{sub}\; (t_1\; t_2)$$

In this case, the reduction sequence must be of the following form, for some mixed term t':

$$\Gamma_{sub}\; (t_1\; t_2) \rightarrow_a^* ((T_2 \Rightarrow t')\; T_2) \rightarrow_a t' \rightarrow_a^* T$$

where

$$\begin{array}{lll} 1. & \Gamma_{sub}\; t_1 & \rightarrow_a^* \quad T_2 \Rightarrow t' \\ 2. & t' & \rightarrow_a^* \quad T \\ 3. & \Gamma_{sub}\; t_2 & \rightarrow_a^* \quad T_2 \end{array}$$

Combining (1) and (2) using transitivity and Lemma 14, we get

$$4.\; \Gamma_{sub}\; t_1 \rightarrow_a^* T_2 \Rightarrow T$$

Now we can apply the induction hypothesis to (3) and (4), which have shorter length than the original reduction sequence. This is because the original reduction sequence has one additional $\rightarrow_a$-step, namely from $((T_2 \Rightarrow t')\; T_2)$ to t'; and Lemma 14 preserves the number of steps in the sequence. This gives us the premises of the following inference, which suffices to complete this case:

$$\frac{\Gamma \vdash t_1 : T_2 \Rightarrow T \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\; t_2 : T}$$

Case:

$$\Gamma_{sub}\; (\lambda x : T'.\, t')$$

In this case, the reduction sequence must be of the following form:

$$\Gamma_{sub}\,(\lambda x : T'.\, t') \to_a (T' \Rightarrow [T'/x]\Gamma_{sub}\, t') \to_a^* (T' \Rightarrow T'')$$

where

$$[T'/x]\Gamma_{sub}\, t' \to_a^* T''$$

Now we can apply the induction hypothesis on $[T'/x]\Gamma_{sub}\ t' \to_a^* T''$ and get the premise of the following inference, which suffices to complete this case:

$$\frac{\Gamma, x : T' \vdash t' : T''}{\Gamma \vdash \lambda x : T'.\, t' : T' \Rightarrow T''}$$

End Proof(⇐).

A.1 Proofs of lemmas 12, 13 and 14:

Proof of Lemma 12. First assume $t_1 \to_a^* T$. We must prove that $t_1\ t_2 \to_a^* T\ t_2$. The proof is by induction on the number of single steps in the derivation of $t_1 \to_a^* T$.

Base Case: There are no single steps. This means that $t_1 = T$. Then we get, $t_1\ t_2 = Tt_2$, which is sufficient for this case.
Step Case: There is at least one single step. Assume that we take a single step from t_1 to t_1' and zero or more steps from t_1' to T as follows:

1. $t_1 \to_a t_1' \to_a^* T$

Suppose that the context used for $t_1 \to_a t_1'$ is E_a and the redex R. Then we can reduce $t_1\ t_2$ using context $E_a[R]\ t_2$, which is a legal abstract evaluation context, as follows:

2. $t_1\ t_2 \to_a t_1'\ t_2$

Now, we can apply the induction hypothesis on $t_1' \to_a^* T$, which is shorter than the original reduction sequence in (1), and get:

3. $t_1'\ t_2 \to_a^* T\ t_2$

Combining (2) and (3) using transitivity, we get:

$$t_1\ t_2 \to_a^* T\ t_2$$

End Proof.

Proof of Lemma 13. The proof is very similar to that of Lemma 12, except for some variation in the context in the step case. So we will just consider the step case.

Step Case: There is at least one single step. Assume that we take a single step from t_2 to t_2' and zero or more steps from t_2' to T as follows:

1. $t_2 \to_a t_2' \to_a^* T$

Suppose that the context used for $t_2 \rightarrow_a t_2'$ is E_a and the redex R. Then we can reduce $t_1\ t_2$ using the context $t_1\ E_a[R]$, which is a legal abstract evaluation context, as follows:

2. $t_1\ t_2 \rightarrow_a t_1\ t_2'$

Now, we can apply the induction hypothesis on $t_2' \rightarrow_a^* T$, which is shorter than the original reduction sequence in (1), and get:

3. $t_1\ t_2' \rightarrow_a^* t_1\ T$

Combining (2) and (3) using transitivity, we get:

$$t_1\ t_2 \rightarrow_a^* t_1\ T$$

End Proof.

Proof of Lemma 14. First we assume $t \rightarrow_a^* T'$. We must prove that $T \Rightarrow t \rightarrow_a^* T \Rightarrow T'$. The proof is by induction on the number of single steps in the derivation of $t \rightarrow_a^* T'$.

Base Case: There are no single steps. This means that $t = T'$. Then we get, $T \Rightarrow t = T \Rightarrow T'$, which is sufficient for this case.

Step Case: There is at least one single step. Assume that we take a single step from t to t' and zero or more steps from t' to T' as follows:

1.$t \rightarrow_a t' \rightarrow_a^* T'$

Suppose that the context used for $t \rightarrow_a t'$ is E_a and the redex R. Then we can reduce $T \Rightarrow t$ using the context $T \Rightarrow E_a[R]$, which is a legal abstract evaluation context, as follows:

2. $T \Rightarrow t \rightarrow_a T \Rightarrow t'$

Now, we can apply the induction hypothesis on $t' \rightarrow_a^* T'$, which is shorter than the original reduction sequence in (1), and get:

3. $T \Rightarrow t' \rightarrow_a^* T \Rightarrow T'$

Combining (2) and (3) using transitivity, we get:

$$T \Rightarrow t \rightarrow_a^* T \Rightarrow T'$$

End Proof.

B Proof of Theorem 10

Proof. The proof proceeds by constructing function M mapping abstract STLC-inf steps $t_1 \rightarrow_a t_2$ to abstract STLC steps $t_1' \rightarrow_a t_2'$ such that $|t_1'| = t_1$ and $|t_2'| = t_2$.

Consider the reduction sequence corresponding to the reduction $t \rightarrow_a^* T$:

$$t_1 \rightarrow_a t_2 \rightarrow_a \ldots \rightarrow_a t_n \rightarrow_a T$$

We can label the reduction steps according to the substitutions performed by the reduction. We assume these are idempotent substitutions. Also, let σ^{n+1} be the substitution mapping the free type

$$
\begin{array}{lcl}
\hat{\sigma} \star \lambda x.m & = & \lambda x : \hat{\sigma}(x).(\hat{\sigma} \star m) \\
\hat{\sigma} \star t_1\ t_2 & = & (\hat{\sigma} \star t_1)\ (\hat{\sigma} \star t_2) \\
\hat{\sigma} \star T \Rightarrow m & = & \hat{\sigma}(T) \Rightarrow (\hat{\sigma} \star m) \\
\hat{\sigma} \star x & = & x \\
\hat{\sigma} \star f & = & f \\
\hat{\sigma} \star a & = & a \\
\hat{\sigma} \star \alpha & = & \hat{\sigma}T
\end{array}
$$

Figure 13 Definition of annotation $\hat{\sigma} \star t$

variables mentioned in the statement of the theorem to the given STLC types $T_1, \ldots, T_n$. We denote the composition of substitutions $\hat{\sigma} = \sigma^1; \sigma^2; \ldots; \sigma^{n+1}$. The domain of the composed substitution $\hat{\sigma}$ includes both term variables x and type variables α_i. The range consists of STLC types.

$$t_1 \xrightarrow{\sigma^1}_a t_2 \xrightarrow{\sigma^2}_a \ldots \rightarrow_a t_n \xrightarrow{\sigma^n}_a T$$

The individual σ^i are determined by the form of the a-step, as follows:

$$
\begin{array}{rcllcl}
E_a[\lambda x.m] & \rightarrow_a & E_a[\alpha_x \Rightarrow [\alpha_x/x]m] & \Longrightarrow & \sigma = \{x \mapsto \alpha_x\} \\
E_a[(T_1 \rightarrow_a m)T_2] & \rightarrow_a & \sigma(E_a[m]) & \Longrightarrow & \sigma = mgu(T_1, T_2) \\
E_a[\alpha_i T] & \rightarrow_a & E_a[\alpha_j] & \Longrightarrow & \sigma = \{\alpha_i \mapsto (T \Rightarrow \alpha_j)\} \\
E_a[f] & \rightarrow_a & E_a[A \Rightarrow A] & \Longrightarrow & \sigma = \{\} \\
E_a[a] & \rightarrow_a & E_a[A] & \Longrightarrow & \sigma = \{\}
\end{array}
$$

Given substitution $\hat{\sigma}$ and a term t we define a *annotation* mapping $\hat{\sigma} \star t$ in Figure 13 that converts an inference term into a STLC term with type variables. The extension of the annotation mapping to contexts is direct, and is omitted here.

Define M to be a mapping between from abstract STLC-inf steps to STLC abstract reduction steps using the substitution labels σ_i:

$$M(E_a[t_i] \xrightarrow{\sigma}_a E_a[t_{i+1}]) = (\hat{\sigma} \star E_a[t_i]) \xrightarrow{\sigma}_a (\hat{\sigma} \star E_a[t_{i+1}])$$

We case split on the form of the abstract inference reduction.

Case: *a(a)* step. By definition of M, $E_a[a] \rightarrow_a E_a[A]$ maps to the STLC reduction $\hat{\sigma} \star E_a[a] \rightarrow_a \hat{\sigma} \star E_a[A]$. The term a and type A are unchanged under annotation resulting in the valid STLC abstract reduction $(\hat{\sigma} \star E_a)[a] \rightarrow_a (\hat{\sigma} \star E_a)[A]$.

Case: *a(f)* step. Same as *a(a)* case.

Case: $a(\lambda)$ step. $M(E_a[\lambda x.m] \rightarrow_a E_a[\alpha_i \Rightarrow [\alpha_i/x]m])$ unfolds to $(\hat{\sigma} \star E_a[\lambda x.m]) \rightarrow_a (\hat{\sigma} \star E_a[\alpha_i \Rightarrow [\alpha_i/x]m])$. Unfolding the definition of $\star$ twice yields $(\hat{\sigma} \star E_a)[\lambda x : \hat{\sigma}(x).\hat{\sigma} \star m] \rightarrow_a (\hat{\sigma} \star E_a)[\hat{\sigma}(\alpha_i) \Rightarrow \hat{\sigma} \star ([\alpha_i/x]m)]$. By the definition of the labeled transition relation, $\hat{\sigma}$ contains the substitution $\sigma_i = x \mapsto \alpha_i$. As x is not free in the range of the composed substitution $\hat{\sigma}$, $\hat{\sigma}(x) = \hat{\sigma}(\alpha_i)$, and $(\hat{\sigma} \star E_a)[\lambda x : \hat{\sigma}(\alpha_i).\hat{\sigma} \star m] \rightarrow_a (\hat{\sigma} \star E_a)[\hat{\sigma}(\alpha_i) \Rightarrow \hat{\sigma} \star ([\alpha_i/x]m)]$. Because $\hat{\sigma}$ contains $x \mapsto \alpha_i$, we can distribute the annotation operation over subsitution $[\alpha_i/x]m$, so $\hat{\sigma} \star ([\alpha_i/x])m = [\hat{\sigma}(\alpha_i)/x](\hat{\sigma} \star m)$. Applying this fact yields the the well-formed STLC reduction $(\hat{\sigma} \star E_a)[\lambda x : \hat{\sigma}(\alpha_i).(\hat{\sigma} \star m)] \rightarrow_a (\hat{\sigma} \star E_a)[\hat{\sigma}(\alpha_i) \Rightarrow [\hat{\sigma}(\alpha_i)/x](\hat{\sigma} \star m)]$.

Case: $a(\beta)$ step. $M(E_a[(T_1 \Rightarrow m)T_2] \rightarrow_a \sigma_i(E_a[m]))$ unfolds to $\hat{\sigma} \star E_a[(T_1 \Rightarrow m)T_2] \rightarrow_a \hat{\sigma} \star (\sigma_i(E_a[m]))$. By definition of $\star$, we get $(\hat{\sigma} \star E_a)[\hat{\sigma} \star (T_1 \Rightarrow m)\ T_2] \rightarrow_a (\hat{\sigma} \star (\sigma_i(E_a)))[\hat{\sigma} \star (\sigma_i(m))]$. Applying the definition of $\star$ to the LHS of the reduction yields $(\hat{\sigma} \star E_a)[(\hat{\sigma}(T_1) \Rightarrow$

$(\hat{\sigma} \star m))\ \hat{\sigma}(T_2)] \rightarrow_a (\hat{\sigma} \star (\sigma_i(E_a)))[\hat{\sigma} \star \sigma_i(m)]$. By the definition of *a*(β), $\hat{\sigma}$ includes the substitution $\sigma_i = mgu(T_1, T2)$. Most general unifiers are closed under composition of substitutions, so $\hat{\sigma}(T_1) \equiv \hat{\sigma}(T_2)$. Thus, we have $(\hat{\sigma} \star E_a)[(\hat{\sigma}(T_1) \Rightarrow (\hat{\sigma} \star m))\ \hat{\sigma}(T_1)] \rightarrow_a (\hat{\sigma} \star (\sigma_i(E_a)))[\hat{\sigma} \star \sigma_i(m)]$. As σ_i is a mgu, that is contained within $\hat{\sigma}$, we can compose the two $\hat{\sigma}(\sigma_i(t)) = \hat{\sigma}(t)$, yielding the valid STLC reduction $\hat{\sigma} \star E_a[(\hat{\sigma}(T_1) \Rightarrow \hat{\sigma} \star m)\hat{\sigma}(T_1)] \rightarrow_a \hat{\sigma} \star E_a[\hat{\sigma} \star m]$

Case: *a*(*gen*) step. $M(E_a[\alpha_i\ T] \rightarrow_a E_a[\alpha_j])$ unfolds to $\hat{\sigma} \star E_a[\alpha_i\ T] \rightarrow_a \hat{\sigma} \star E_a[\alpha_j]$, then in turn $(\hat{\sigma} \star E_a)[\hat{\sigma} \star \alpha_i\ T] \rightarrow_a (\hat{\sigma} \star E_a)[\hat{\sigma} \star \alpha_j]$. Unfolding $\hat{\sigma} \star (\alpha_i\ T)$, we get $(\hat{\sigma} \star E_a)[\hat{\sigma} \star \alpha_i\ \hat{\sigma} \star T] \rightarrow_a (\hat{\sigma} \star E_a)[\hat{\sigma} \star \alpha_j]$. By definition, $\hat{\sigma}$ contains the substitution $\alpha_i \mapsto T$; applying that substitution within the LHS of the reduction step yields the well-formed STLC reduction $(\hat{\sigma} \star E_a)[(\hat{\sigma} \star T \Rightarrow (\hat{\sigma} \star \alpha_j))\ (\hat{\sigma} \star T)] \rightarrow_a (\hat{\sigma} \star E_a)[\hat{\sigma} \star \alpha_j]$.

The function M maps each step in an abstract STLC-inf reduction to a suitably equivalent abstract STLC step. Each step in the STLC derivation corresponds to a step with annotated STLC-inf terms, and by Lemma 15 the erasure of an annotated term yields the original term. **End Proof**.

▶ **Lemma 15.**

$\forall t, \hat{\sigma}.|\hat{\sigma} \star t| = t.$

Proof. Straightforward induction on definition of |t|. **Omitted**.