A REWRITING VIEW OF SIMPLE TYPING

AARON STUMP, GARRIN KIMMELL, HANS ZANTEMA, AND RUBA EL HAJ OMAR

Computer Science, The University of Iowa
e-mail address: astump@acm.org

Computer Science, The University of Iowa
e-mail address: garrin-kimmell@uiowa.edu

Department of Computer Science, TU Eindhoven, The Netherlands; and Institute for Computing and Information Sciences, Radboud University, The Netherlands
e-mail address: h.zantema@tue.nl

Computer Science, The University of Iowa
e-mail address: roba-elhajomar@uiowa.edu

Abstract. This paper shows how a recently developed view of typing as small-step abstract reduction, due to Kuan, MacQueen, and Findler, can be used to recast the development of simple type theory from a rewriting perspective. We show how standard metatheoretic results can be proved in a completely new way, using the rewriting view of simple typing. These metatheoretic results include Type Preservation and Progress for several variants of simply typed lambda calculus, including a version with a fixed-point combinator, a Curry-style system with type-inference, and a simply typed combinatory language with uniform syntax. We also include a new proof of normalization of simply typed lambda calculus, based on the rewriting approach.

1. Introduction

This paper develops a significant part of the theory of simple types based on a recently introduced rewriting approach to typing. The idea of viewing typing as a small-step abstract reduction relation was proposed by Kuan, MacQueen, and Findler in 2007, and explored also by Ellison, Şerbănuţă, and Roşu [9, 7, 10]. These works sought to use rewrite systems to specify typing in a finer-grained way than usual type systems. Our motivation is more foundational: we seek to apply powerful tools from term-rewriting theory to prove standard metatheoretic properties of type systems, in particular type safety and normalization for simply typable terms. The goal is to develop new methods which could provide a different

1998 ACM Subject Classification: D.3.1 Formal Definitions and Theory.
Key words and phrases: Term rewriting, Type Safety, Confluence.
This work was partially supported by the U.S. National Science Foundation, contract CCF-0910510, as part of the Trellys project.

perspective on familiar type systems, and perhaps yield new results for more advanced type systems.

Our focus in this paper is simple type systems, where the central typing construct is the function type $T \Rightarrow T'$. On the rewriting view, we will view such types as abstractions of functions, and incrementally rewrite (typable) functions to such function types, using an abstract (small-step) reduction relation. As we will see, this viewpoint allows us to treat the usual concrete reduction relation and this new abstract reduction relation together, simply by taking the set-theoretic union of the two relations. Our first main result is to show confluence of the combined reduction relation for typable terms, defined as terms which reduce (using abstract or concrete steps) to a type. This notion of typability strictly generalizes the usual notion, as we will discuss below. To prove confluence we use a new theorem which we dub the Combined Confluence Theorem, developed in the context of abstract reduction systems. From confluence for typable terms, we then derive a generalized form of type safety. We show how this general approach scales to several variants of simply typed lambda calculus. Along the way, we will see an example of how automated analysis tools for term-rewriting systems can partially automated the proof of type preservation for a simply typed language. We then develop our second main result, which is a proof of normalization for the simply typed lambda calculus, based on the rewriting approach. This proof has several novel features, which shed light both on the rewriting approach to typing, and standard proofs of normalization based on reducibility.

This paper expands in several important ways on a previous paper of Stump, Kimmell, and El Haj Omar, which was presented at RTA 2011 [14]:

- We present the Combined Confluence Theorem, and contribute both a self-contained proof and a proof based on decreasing diagrams. The original RTA '11 paper based all the results directly on decreasing diagrams, and did not abstract out the essential characteristics of the local diagrams which are necessary in this setting to achieve confluence.
- We expanded our notion of typable term. In the original paper, we took typability to mean abstract reduction to a type. We have now relaxed this, so that typability just means reduction to a type, using steps which each may be either abstract or concrete. This is significant generalization of the previous development, and requires a reworking of parts of both the confluence proof and the proof of Progress.
- We have added the proof of normalization for simply typable terms. This is a new result, not present in the RTA paper.

As Zantema had a substantial contribution to these extensions, he was added as an author.

The remainder of the article is organized as follows. Section 2 provides a brief introduction to abstract reduction systems as used later in the paper. Section 3 gives a standard presentation of the simply typed lambda calculus along with the fundamental metatheoretic properties. Section 4 recasts the simply typed lambda calculus static and operational semantics within framework of abstract reduction systems, and Section 5 shows that type preservation in this system is a consequence of confluence. Section 6 provides a new theorem of confluence for abstract reduction systems, and Section 7 applies that theorem to the abstract reduction system presentation of the simply typed lambda calculus. Section 8 extends the metatheoretic analysis with progress and type safety results. Section 9 applies the techniques to the study of variants of the simply typed lambda calculus. Section 10 applies a rewriting approach to prove the normalization of well-typed simply typed lambda calculus terms. We conclude and identify future directions in Section 11.

2. Rewriting Preliminaries

In this section we collect some basic properties in the setting of abstract reductions systems. That is, we consider relations $\to$ being a subset of $A \times A$ for some arbitrary set A.

We write $\cdot$ for relation composition, and inductively define $\to^0 = id$ (the identity) and $\to^n = \to^{n-1} \cdot \to$ for $n > 0$. As usual, for a relation $\to$ we write $\leftarrow$ for its reverse, $\to^=$ for its reflexive closure (zero or one times), $\to^+ = \bigcup_{i=1}^{\infty} \to^i$ for its transitive closure (one or more times), and $\to^* = \bigcup_{i=0}^{\infty} \to^i$ for its transitive reflexive closure (zero or more times).

A relation $\to$ is said to

- be *confluent* (Church Rosser, $CR(\to)$) if $\leftarrow^* \cdot \to^* \subseteq \to^* \cdot \leftarrow^*$,
- be *locally confluent* (Weak Church Rosser, $WCR(\to)$) if $\leftarrow \cdot \to \subseteq \to^* \cdot \leftarrow^*$,
- have the *diamond property* ($\diamond(\to)$) if $\leftarrow \cdot \to \subseteq \to^= \cdot \leftarrow^=$,
- be *deterministic* ($\det(\to)$) if $\leftarrow \cdot \to \subseteq id$.

It is well-known and easy to see that $\det(\to) \Rightarrow \diamond(\to) \Rightarrow CR(\to) \Rightarrow WCR(\to)$. We will need some further more advanced results from rewriting theory, presented in Section **??** below.

3. A Standard Presentation of Simple Typing

In this section, we summarize a standard presentation of the simply typed lambda calculus, including syntax and semantics, and statements of the basic metatheoretic properties of Type Preservation and Progress. Sections 4 and following will recapitulate this development in detail, from the rewriting perspective. 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.

3.1. 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:

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

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. Following standard usage, terms of the form $(\lambda x : T.t)\ v$ or $f\ a$ are called redexes.

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

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

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

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

3.2. **Basic Metatheory.** The main theorem relating the reduction relation $\rightarrow$ and typing is **Type Preservation**, which states:

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

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:

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

One also typically proves **Progress**:

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

Here, the notation $t \not\rightarrow$ means $\forall t'.\ \neg(t \rightarrow t')$; i.e., t is a normal form. Normal forms which are not values are called *stuck* terms. An example is $f\ f$. Combining Type Preservation and Progress 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:

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

This is proved by induction on the length of the reduction sequence from t to t'. As already noted, without constants (f and a here), this result is not so interesting for STLC, since it follows already by simpler reasoning: reduction cannot introduce new free variables, so t' must be closed; and it is then easy to prove that closed normal forms are λ-abstractions, and hence values by definition.

4. Simple Typing as Abstract Reduction

In this section, we see how to view a type-computation (also called type-synthesis) system for STLC as an abstract operational semantics. Section 5 will prove type preservation as a form of confluence for the combination of the abstract and the standard concrete 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

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

Figure 3: Syntax for STLC using mixed terms

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

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

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

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

Figure 4: Abstract and concrete operational semantics for STLC

concrete ($\to_c$) and abstract ($\to_a$) reduction. As above, we will refer to any term of the form displayed in context on the left hand side of the conclusion of a rule as a redex. 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 operational semantics:

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

The final result is a type T. Indeed, using the standard typing rules of Section 3.1, 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 4.1 (Relation with Standard Typing)**.** *For standard terms t, we have $x_1 : T_1, \cdots, x_n : T_n \vdash t : T$ iff $[T_1/x_1, \cdots, T_n/x_n]t \rightarrow_a^* T$.*

Proof ($\Rightarrow$). 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 (see the Appendix for the proofs):

Lemma 4.2. *If $t_1 \rightarrow_a^* T$, then $t_1 \ t_2 \rightarrow_a^* T \ t_2$ using the same number of steps.*

Lemma 4.3. *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:

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

Now we need to show that

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

We will need the following lemma to construct the reduction sequence below (see the Appendix for the proof):

Lemma 4.4. *If $t \to_a^* T'$, then $T \Rightarrow t \to_a^* T \Rightarrow T'$ using the same number of steps.*

Now, by applying one abstract(λ) and Lemma 4.4 we get:

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

This requires the fact that $\Gamma_{sub}\ [T/x] = [T/x]\Gamma_{sub}$, which holds because $x \notin \mathit{dom}(\Gamma_{sub})$ 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 $[T_1/x_1, \cdots, T_n/x_n]t \to_a^* T$ and prove $x_1 : T_1, \cdots, x_n : T_n \vdash t : T$ by induction on the number of single steps in $\to_a^*$.

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

$$\Gamma_{sub}\ f$$

The only possible step is $f \to_a A \Rightarrow A$, 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':

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

where

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

Combining (1) and (2) using transitivity and Lemma 4.4, 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 4.4 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:

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

where

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

Now we can apply the induction hypothesis on $[T'/x]\Gamma_{sub}\ t' \rightarrow_a^* T''$ 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($\Leftarrow$).

Theorem 4.5 (Termination of Abstract Reduction)**.** *The relation $\rightarrow_a$ is terminating.*

Proof. Whenever $m \rightarrow_a m'$, 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.**

5. Type Preservation as Confluence

We will now see how a generalized form of type preservation can be seen as a simple consequence of confluence of combined reduction. If we translate our standard statement of Type Preservation (at the start of Section 3.2) so that it uses abstract reduction instead of the usual typing relation, we have the following statement:

Theorem 5.1 (Type Preservation)**.** *If $m \rightarrow_a^* T$ and $m \rightarrow_c m'$, then $m' \rightarrow_a^* T$.*

Our goal, of course, is to prove this directly, which we will do in the course of Section 7 just below. But more interestingly, we will explore a natural generalization of this property, based on confluence. The relation $\rightarrow_{ca}$ 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 $m \rightarrow_a^* T$ for some type T. We will make use here of the standard notion of confluence of an element with respect to a binary relation $\rightarrow$: m is confluent (with respect to $\rightarrow$) iff for all m_1 and m_2 such that $m_2 \leftarrow^* m \rightarrow^* m_1$, there exists $\hat{m}$ such that $m_1 \rightarrow^* \hat{m} \leftarrow^* m_2$. In the next section, we prove the following result:

Theorem 5.2 (Confluence of Combined Reduction)**.** *Every typable mixed term is confluent with respect to the reduction relation $\rightarrow_{ca}$.*

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 5.3 (Generalized Type Preservation)**.** *If $m \rightarrow_{ca}^* T$ and $m \rightarrow_c^* m'$, then $m' \rightarrow_{ca}^* T$.*

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. We call this generalized type preservation, since we use *ca*-reduction instead of just *a*-reduction to correspond to typing.

Using this notion of generalize type preservation, we will see in Section 8 that we can prove a correspondingly generalized form of type safety. This suggests that confluence, which is at the heart of generalized type preservation, could play an essential role as part of a general methodology for analyzing type systems. Confluence is a stronger property, of course, and may be more difficult to prove by hand. But as we will see in Section 9.3, it may be possible to make use of automated confluence provers to automate part of this proof. This would then make proving type safety based on generalized type preservation (as a direct corollary of confluence) an attractive approach, compared to proving just type preservation by hand.

6. The Main Theorem for Confluence of Combined Reduction

In this section, we apply techniques from the theory of abstract reduction systems (ARS's) to prove confluence of the combined abstract and concrete reduction relation. First, we develop a new theorem (Theorem 6.2) which can be used to show confluence for all the systems considered in this paper. In the next section we will see how to apply this theorem for the case of STLC.

A crucial issue in this paper is to conclude confluence of a relation $\rightarrow$ that is defined to be the union of a confluent relation $\rightarrow_1$ and a relation $\rightarrow_2$ satisfying some extra properties. It is quite obvious that without further requirements this is not the case. For instance, if $\rightarrow_1 = \{(a, b)\}$ and $\rightarrow_2 = \{(a, c)\}$ then both $\rightarrow_1$ and $\rightarrow_2$ are confluent while $\rightarrow_1 \cup \rightarrow_2$ is not.

A standard result in this is area is the theorem on *decreasing diagrams* [17], which applies on any set of relations equipped with a well-founded order. Here we formulate the version on only two relations $\rightarrow_1$ and $\rightarrow_2$ with well-founded order $\rightarrow_1 < \rightarrow_2$.

Theorem 6.1 (decreasing diagrams [17])**.** *Let $\rightarrow_1$ and $\rightarrow_2$ be relations such that*

- $\diamond(\rightarrow_1)$,
- $\leftarrow_2 \cdot \rightarrow_2 \subseteq \rightarrow_1^* \cdot \rightarrow_2^= \cdot \rightarrow_1^* \cdot \leftarrow_1^* \cdot \leftarrow_2^= \cdot \leftarrow_1^*$,

- $\leftarrow_1 \cdot \rightarrow_2 \subseteq \rightarrow_1^* \cdot \rightarrow_2^= \cdot \rightarrow_1^* \cdot \leftarrow_1^*$.

Then $CR(\rightarrow_1 \cup \rightarrow_2)$.

However, for our goals this theorem does not apply directly: typically our basic relation $\rightarrow_1$ is confluent, but does not satisfy the diamond property. The following theorem is sufficient to serve for our goals.

Theorem 6.2 (Combined Confluence)**.** *Let* $\rightarrow_1$ *and* $\rightarrow_2$ *be relations such that*

- $CR(\rightarrow_1)$,
- $\leftarrow_2 \cdot \rightarrow_2 \subseteq \rightarrow_1^* \cdot \rightarrow_2^= \cdot \rightarrow_1^* \cdot \leftarrow_1^* \cdot \leftarrow_2^= \cdot \leftarrow_1^*$,
- $\leftarrow_1 \cdot \rightarrow_2 \subseteq \rightarrow_2^= \cdot \rightarrow_1^* \cdot \leftarrow_1^*$.

Then $CR(\rightarrow_1 \cup \rightarrow_2)$.

Proof. We give two proofs of this theorem: one using Theorem 6.1 and one self-contained. For both approaches we first prove by induction on n that

$$\leftarrow_1^n \cdot \rightarrow_2^= \subseteq \rightarrow_2^= \cdot \rightarrow_1^* \cdot \leftarrow_1^*$$

for all $n \geq 0$. For $n = 0$ this is trivial, for $n > 0$ we have

$$\begin{array}{rcll} \leftarrow_1^n \cdot \rightarrow_2^= & = & \leftarrow_1^{n-1} \cdot \leftarrow_1 \cdot \rightarrow_2^= & \\ & \subseteq & \leftarrow_1^{n-1} \cdot \rightarrow_2^= \cdot \rightarrow_1^* \cdot \leftarrow_1^* & \\ & \subseteq & \rightarrow_2^= \cdot \rightarrow_1^* \cdot \leftarrow_1^* \cdot \rightarrow_1^* \cdot \leftarrow_1^* & \text{(induction hypothesis)} \\ & \subseteq & \rightarrow_2^= \cdot \rightarrow_1^* \cdot \leftarrow_1^* & (CR(\rightarrow_1)), \end{array}$$

by which we have proved

$$\leftarrow_1^* \cdot \rightarrow_2^= \subseteq \rightarrow_2^= \cdot \rightarrow_1^* \cdot \leftarrow_1^* .$$

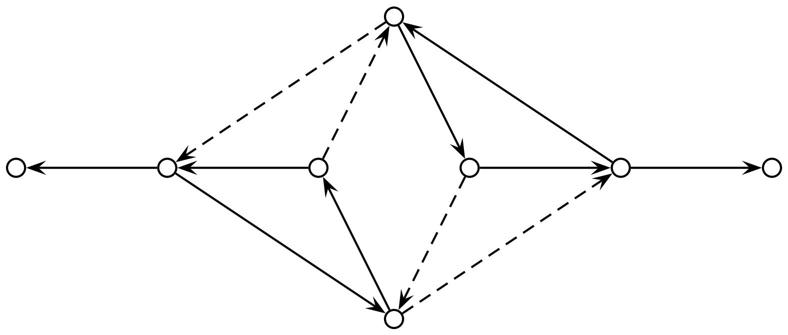
Using this property, all conditions of Theorem 6.1 hold for $\rightarrow_1^*$ and $\rightarrow_2$; note that $\rightarrow_1^{**} = \rightarrow_1^*$ and $CR(\rightarrow_1)$ coincides with $\diamond(\rightarrow_1^*)$. So by Theorem 6.1 we conclude $CR(\rightarrow_1^* \cup \rightarrow_2)$ from which $CR(\rightarrow_1 \cup \rightarrow_2)$ follows. This concludes the first proof of Theorem 6.2.

For the self-contained proof consider an arbitrary conversion with respect to $\rightarrow_1 \cup \rightarrow_2$. This conversion can be represented by a string over the symbols a, A, b, B, representing $\leftarrow_1^*, \rightarrow_1^*, \leftarrow_2, \rightarrow_2$, respectively. For instance, a conversion $\rightarrow_2 \cdot \leftarrow_1 \cdot \leftarrow_1 \cdot \rightarrow_2 \cdot \rightarrow_1$ is represented by the string $BaBA$. Using $CR(\rightarrow_1)$ every peak of the shape $\leftarrow_1^* \cdot \rightarrow_1^*$ may be replaced by a valley $\rightarrow_1^* \cdot \leftarrow_1^*$, that is, a substring aA is replaced by Aa. Since $\leftarrow_2 \cdot \rightarrow_2 \subseteq \rightarrow_1^* \cdot \rightarrow_2^= \cdot \rightarrow_1^* \cdot \leftarrow_1^* \cdot \leftarrow_2^= \cdot \leftarrow_1^*$, every peak of the shape $\leftarrow_2 \cdot \rightarrow_2$ may be replaced by a corresponding valley, that is, a substring bB is replaced by $A[B]Aa[b]a$, in which $[-]$ means that the symbol inside may be removed. Using $\leftarrow_1^* \cdot \rightarrow_2^= \subseteq \rightarrow_2^= \cdot \rightarrow_1^* \cdot \leftarrow_1^*$, every substring aB may be replaced by $[B]Aa$, and by symmetry, every substring bA may be replaced by $Aa[b]$. Summarizing, the process of peak completion can be described by the string rewrite system S consisting of the following six rules

$$\begin{array}{rcl} aA & \rightarrow & Aa \\ bB & \rightarrow & ABAaba \\ aB & \rightarrow & BAa \\ bA & \rightarrow & Aab \\ b & \rightarrow & \epsilon \\ B & \rightarrow & \epsilon, \end{array}$$

in which ϵ stands for the empty string. The key argument now is the observation that S is terminating. This is easily proved automatically by any current termination tool. For instance, TORPA 1.4 finds a proof that is easily checked by hand: first by choosing the polynomial interpretation in which b is interpreted by multiply by 10, B by the successor function, and the other symbols by the identity, the second rule and the last two rules are removed. Next the remaining three rules are proved to be terminating by the recursive path order with respect to $b > a > A > B$, concluding the termination proof. Hence the string representing the original conversion has a normal form. Due to the shape of the rules, this normal form is of the shape $(A + B)^*(a + b)^*$, so represents a valley conversion. Since by every rewrite step the conversion is replaced by another conversion of the same two elements, we have proved that these two elements admit a valley conversion, exactly what we had to prove for proving $CR(\rightarrow_1 \cup \rightarrow_2)$. **End Proof.**

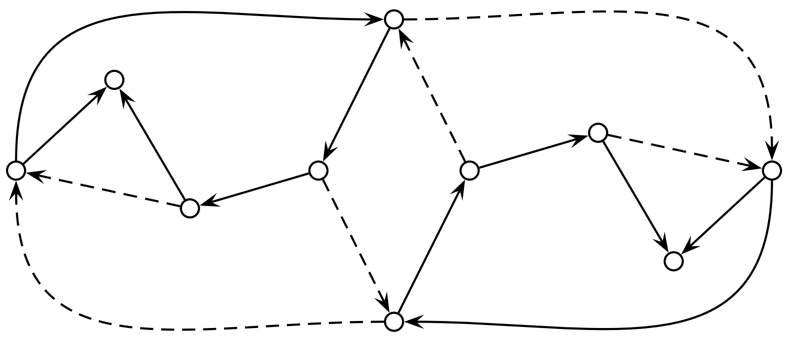
Theorem 6.2 would have more interesting applications if the second requirement would be replaced by $CR(\rightarrow_2)$. Then it could be applied for instances where $\rightarrow_2$ corresponds to β reduction. Unfortunately, this desired variant does not hold, as is shown by the following relation on 8 elements. In the picture $\rightarrow_1$ steps are denoted by dashed arrows and $\rightarrow_2$ steps are denoted by solid arrows.



Confluence of $\rightarrow_1 \cup \rightarrow_2$ does not hold since the leftmost and rightmost elements are convertible normal forms. Confluence of $\rightarrow_1$ and $\rightarrow_2$ separately are easily checked, $\rightarrow_1$ is even deterministic. Also the third condition of Theorem 6.2 holds, even stronger: $\leftarrow_1 \cdot \rightarrow_2 \subseteq \rightarrow_1 \cup (\rightarrow_2 \cdot \leftarrow_1)$.

After first finding more complicated and less symmetrical examples by hand, this example was found by expressing the properties of a counter example in a SAT formula, let a SAT solver prove satisfiability, and interpret the found satisfying assignment. The same exercise on 7 elements yielded an unsatisfiable formula, showing that no the smaller example exists than the given one.

In this example the relation $\rightarrow_2$ is clearly non-terminating. A next question is whether Theorem 6.2 would hold in case the second requirement was replaced by the combination of confluence and termination of $\rightarrow_2$. If so, then it could be applied for instances where $\rightarrow_2$ corresponds to β reduction in a simply typed setting. Unfortunately, also this variant does not hold, as is shown by the following relation on 10 elements. Again $\rightarrow_1$ steps are denoted by dashed arrows and $\rightarrow_2$ steps are denoted by solid arrows.



For this second example again there are two convertible normal forms, so the union is not confluent, and both $\rightarrow_1$ and $\rightarrow_2$ are both confluent and terminating; $\rightarrow_1$ is even deterministic. Also the third condition of Theorem 6.2 is easily checked, even stronger: $\leftarrow_1 \cdot \rightarrow_2 \subseteq (\rightarrow_1 \cup \rightarrow_2) \cdot \leftarrow_1^=$. Again this example was found using a SAT solver. A direct encoding of the example to be looked for run out of resources. However, by adding a symmetry requirement similar to what was observed on the first example, the SAT solver yielded a satisfying assignment that could be interpreted as a valid example. The example given above was obtained from this after removing some redundant arrows. Independently, Bertram Felgenhauer found an example that could be simplified to exactly the same example as given here.

7. Applying the Combined Confluence Theorem to STLC

We now use the Combined Confluence Theorem (Theorem 6.2) to prove Theorem 5.2. In our prior work, we proved Theorem 5.2 by applying decreasing diagrams directly [14]. Using the Combined Confluence Theorem is more informative, as it pinpoints the exact form required for local diagrams, and also relieves the development from a dependence on the theory of decreasing diagrams, which is rather complex. This could be of value for formalizing the results in this section in a theorem prover, for example.

We will always apply the Combined Confluence Theorem by taking $\rightarrow_a$ as $\rightarrow_1$ and $\rightarrow_c$ as $\rightarrow_2$. So our first obligation to be able to apply the theorem to conclude confluence of $\rightarrow_{ca}$ is to show confluence of $\rightarrow_a$.

Theorem 7.1. *The relation $\rightarrow_a$ is confluent.*

Proof. In fact, we can prove that $\rightarrow_a$ has the diamond property (i.e., $(\leftarrow_a \cdot \rightarrow_a) \subseteq \rightarrow^= \cdot \leftarrow^=$, which is well-known to imply confluence). Suppose $m \rightarrow_a m_1$ and $m \rightarrow_a m_2$. 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 $m \rightarrow_a m_{3-i}$, 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 [15]) 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. This gives the diamond property and thus confluence for $\to_a$. **End proof.**

We return now to applying the Combined Confluence Theorem. Our next obligation is to prove that every cc-peak can be completed to a diagram of the form described in the second condition of the theorem. Since $\to_c$ is deterministic (due to the restrictions inherent in the definition of reduction contexts E_c), all cc-peaks are of the form $m' \leftarrow_c m \to_c m'$, which are completable via $m' \to_c^= m' \leftarrow_c^= m'$, which has a form allowed by the second condition of the Combined Confluence Theorem.

Finally, we must consider ac-peaks, of the form $m_1 \leftarrow_a m \to_a m_2$ $m_1 \leftarrow_a m \to_c m_2$, and show that these can be completed following the pattern in the third condition of the Combined Confluence Theorem. 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 forms of the joining sides are as required by the Combined Confluence Theorem. 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:

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

This is again of the required form. 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 (which is a consequence of Lemma 8.3 below), to conclude that contracting the a-redex in u indeed results in a new value u':

Lemma 7.2. *If $u \to_a^* m$ then m is also a mixed value.*

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

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

The joining valley, which again satisfies the requirements of the Combined Confluence Theorem, is:

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

Second:

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

The joining valley is:

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

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

Figure 5: Crucial local diagram for STLC

Finally, the more complicated case is shown in Figure 5. We are using the fact that $u \to_a^* T_1$, which we deduce as follows. First, since the term at the peak is typable, we know that $u \to_{ca}^* T_1$. This is because reduction cannot eliminate an application from a term, except by reducing that application itself. The only reduction step which can erase a term is $c(\beta)$, and there the erased term must be a value. So we know that $u \to_{ca}^* T_1$, or else the application displayed at the peak could not be eliminated by abstract reduction. Second, we can show that since u is a mixed value, if $u \to_{ca}^* T_1$, then we must have $u \to_a^* T_1$. This result is intuitive, but we defer its proof to Section 8 below (Lemma 8.4), where it falls out naturally during the course of proving Progress.

Considering the diagram in Figure 5, we see that again, the joining steps satisfy the form required by the Combined Confluence Theorem. Note that the bottom-right joining step, from $E_c[[u/x]m]$ to $E_c[[T_1/x]m]$ requires nondeterminism of $\to_a$ (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. This completes the proof that the conditions of the Combined Confluence Theorem are satisfied, which is sufficient then to conclude **Confluence** (Theorem 5.2).

8. Progress and Type Safety

In this section, we complete the basic metatheory for STLC by proving Progress and Type Safety theorems. We will prove both standard and generalized forms of these theorems, corresponding to Type Preservation and Generalized Type Preservation (Theorems 5.1 and 5.3 above).

8.1. **Quasi-Stuck Terms.** We begin by inductively defining the set of **quasi-stuck** terms S, in Figure 6. Also, let us call a quasi-stuck term which is not a value **stuck**. The purpose of these definitions is to generalize a characterization of c-normal standard terms to mixed terms (Lemmas 8.1 and 8.2, proved next), in such a way that we can show that the set of quasi-stuck terms is closed under abstract reduction (Lemma 8.3, proved below). This will allow us to prove that typable quasi-stuck terms must be values (Lemma 8.5), from which we easily obtain the desired main theorems of (Generalized) Progress and Type Safety.

Lemma 8.1. *If m is quasi-stuck, then $m \not\to_c$.*

- Mixed values u are in S.
- Terms of the form $a\ s$ or $A\ s$ are in S if $s \in S$.
- Terms of the form $f\ s$ or $(A \Rightarrow A)\ s$ are in S if $s \in S$ and s is neither a nor A.
- Terms of the form $(\lambda x : T.m)\ s$ or $(T \Rightarrow m)\ s$ are in S if $s \in S$ and s is not a mixed value.
- Terms of the form $s\ s'$ are in S if $s, s' \in S$ and s is not a mixed value.

Figure 6: Inductive definition of the set S of quasi-stuck terms

Proof. The proof is by an easy induction on the form of m, using the definition of quasi-stuck. **End Proof.**

Lemma 8.2. *If standard term t is closed and $t \not\to_c$, then t is quasi-stuck.*

Proof. The proof is by induction on t. If t is a (standard) value it is quasi-stuck, and it cannot be a variable since t is closed. So suppose it is an application $t_1\ t_2$. Since t_1 is closed, t_1 cannot be a variable. We consider now the remaining possibilities. It could be that t_1 is a and t_2 is some other c-normal form. Then by the induction hypothesis, t_2 is quasi-stuck, and t is, too, using the second clause above in the definition of quasi-stuck terms. Next, we could have the situation where t_1 is f, and t_2 is any c-normal form except a. Then by the induction hypothesis, t_2 is quasi-stuck, and t is, too, using the third clause in the definition of quasi-stuck terms. Next, we could have that t_1 is a λ-abstraction, and t_2 is any c-normal form except a standard value. Then by the induction hypothesis, t_2 is quasi-stuck, and it cannot be a mixed value other than a standard value, because t_2 is a standard term. So t is quasi-stuck, too, using the fourth clause . Finally, if t_1 is some application, then by the induction hypothesis, t_1 and t_2 are both quasi-stuck. Since t_1 is not a value, the fifth clause above gives us that t is quasi-stuck. **End Proof.**

Lemma 8.3 (Reduction of Quasi-Stuck Terms)**.** *If m is quasi-stuck, and $m \to_a m'$, then m' is also quasi-stuck. Furthermore, if m is a mixed value, then so is m'; and if m is not a mixed value, then neither is m'.*

Proof. The proof is by induction on the form of m. Suppose m is a mixed value. Then it is easy to see by inspection of the reduction rules that m' must be, too. So suppose m is of the form $a\ s$ or $A\ s$ with $s \in S$. Then either the assumed reduction is of the form $a\ s \to_a A\ s$, or else of the form $a\ s \to_a a\ m''$ or $A\ s \to_a A\ m''$. In the former case, the resulting term is a quasi-stuck non-value. In the latter, we may apply the induction hypothesis to conclude that m'' is quasi-stuck, and hence $a\ m''$ (or $A\ m''$) is a quasi-stuck non-value.

If m is of the form $f\ s$ or $(A \Rightarrow A)\ s$ with $s \in S$ and s not a or A, then either the assumed reduction is of the form $f\ s \to_a (A \Rightarrow A)\ s$ or else $f\ s \to_a f\ m''$ or $(A \Rightarrow A)\ s \to_a (A \Rightarrow A)\ m''$. In the former case, the resulting term is a quasi-stuck non-value, by the third clause of the definition of quasi-stuck terms above. In the latter, if s is not a value, we again use our induction hypothesis to conclude that m'' is a quasi-stuck non-value, and hence not a or A. So m' is a quasi-stuck non-value, too. If s is a value, then so is m'', and reduction cannot turn a value other than a into a or A. So again, m'' has the required form to be a quasi-stuck non-value.

Suppose m is of the form $(\lambda x : T.m'')\ s$ or $(T \Rightarrow m'')\ s$, with $s \in S$ and s not a mixed value. Then either the assumed reduction is of the form $(\lambda x : T.m'')\ s \to_a (T \Rightarrow [T/x]m'')\ s$; or else of the form $(\lambda x : T.m'')\ s \to_a (\lambda x : T.m''')\ s$ or $(T \Rightarrow m'')\ s \to_a (T \Rightarrow m''')\ s$; or else of the form $(\lambda x : T.m'')\ s \to_a (\lambda x : T.m'')\ m'''$ or $(T \Rightarrow m'')\ s \to_a (T \Rightarrow m'')\ m'''$. In the first two cases, the resulting term still has the required form to be a quasi-stuck non-value. In the third case, we know s is not a value by the definition of quasi-stuck terms, so we may use our induction hypothesis to conclude that m''' is a quasi-stuck non-value, which is sufficient to conclude that the resulting term is again stuck.

Finally, suppose m is of the form $m_1\ m_2$, where m_1 is not a mixed value. Then the assumed reduction must be of the form either $m_1\ m_2 \to_a m_1'\ m_2$ or else $m_1\ m_2 \to_a m_1\ m_2'$, for some m_1' with $m_1 \to_a m_1'$, or else some m_2' with $m_2 \to_a m_2'$. This is because, by inspection of the reduction rules, m itself cannot be a redex if m_1 is not a mixed value. In the former case, we may apply the induction hypothesis to conclude that m_1' is a quasi-stuck non-value, and hence so is m'. In the latter, we may apply the induction hypothesis to conclude that m_2' is quasi-stuck, and hence so is m'. **End Proof.**

Lemma 8.4. *If m is quasi-stuck (including the case where m is a closed mixed value), and $m \to_{ca}^* T$, then $m \to_a T$.*

Proof. The proof is by induction on the length of the reduction sequence from m to T. If this length is 0, the result holds, since types are mixed values. So suppose we have $m \to_{ca} m' \to_{ca}^* T$. Since m is quasi-stuck, we have $m \not\to_c$ by Lemma 8.1. So it must be the case that $m \to_a m'$. Since m' is quasi-stuck by Lemma 8.3, we may apply our induction hypothesis to conclude $m' \to_a^* T$, and hence $m \to_a^* T$. **End Proof.**

Lemma 8.5 (Typable Quasi-Stuck Terms)**.** *Suppose m is a closed quasi-stuck term. Suppose further that $m \to_{ca}^* T$. Then m is a mixed value.*

Proof. The proof is similar to the previous one, and proceeds by induction on the length of the reduction sequence from m to T. If this length is 0, the result holds, since types are mixed values. So suppose we have $m \to_{ca} m' \to_{ca}^* T$. Since m is quasi-stuck, we have $m \not\to_c$ by Lemma 8.1. So it must be the case that $m \to_a m'$. We now consider cases on the form of m. If m is a mixed value the result holds. So suppose it is a non-value. Then by Lemma 8.3, m' must also be a quasi-stuck non-value, and we may apply the induction hypothesis to derive a contradiction. **End proof.**

8.2. **Concluding Progress and Type Safety.**

Armed with the concept of quasi-stuck terms and its associated lemmas, we can now obtain the main results of this section.

Theorem 8.6 (Progress)**.** *If standard term t is closed, $t \to_a^* T$, and $t \not\to_c$, then t is a (standard) value.*

Proof. We can obtain this result by applying Lemma 8.5. To do this, we must note that our hypothesis $t \to_a^* T$ implies that lemma's condition of $t \to_{ca}^* T$. Also, we must use Lemma 8.2 to conclude that t is quasi-stuck, from the assumption that t is c-normal. **End Proof.**

Theorem 8.7 (Type Safety)**.** *If standard term t is closed, $t \to_a^* T$, and $t \to_c^* t' \not\to_c$, then t' is a (standard) value.*

Proof. This is a direct corollary of Theorems 5.1 and 8.6. **End proof.**

Theorem 8.8 (Generalized Progress)**.** *If standard term t is closed, $t \to_{ca}^* T$, and $t \not\to_c$, then t is a (standard) value.*

Proof. As for Theorem 8.6, we obtain this result by applying Lemma 8.5, making using also of Lemma 8.2. **End Proof.**

Theorem 8.9 (Generalized Type Safety)**.** *If standard term t is closed, $t \to_{ca}^* T$, and $t \to_c^* t' \not\to_c$, then t' is a (standard) value.*

Proof. This is a direct corollary of Theorems 5.3 and 8.8. **End proof.**

Generalized Type Safety liberalizes the standard notion of Type Safety, by using ca-reduction instead of just a-reduction, to correspond to typing. This is a true generalization, since it applies to terms that are not strictly speaking typable, but which ca-reduce to a type. A simple example is $(\lambda x : A.\ \lambda y : A.y)\ \lambda x : A.x\ x$. Note that the argument term $\lambda x : A.x\ x$ is not simply typable. This term has several ca-reduction sequences, including the following one:

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

Because this term ca-reduces to a type, Generalized Type Safety tells us that its c-normal form, if such exists, is a value. This can, of course, be confirmed for this case, where the c-normal form is just $\lambda y : A.y$.

9. Analyzing Variants of STLC

In this section, we show how the ideas developed above for STLC can be adapted to three variants of STLC: STLC plus a fixed-point combinator, STLC with type inference, and a combinatory version of STLC with uniform syntax. These examples are chosen to illustrate different language features that could (and do) affect the application of the Combined Confluence Theorem. For example, STLC with a fixed-point combinator no longer has the normalization property for concrete reduction, and the combinatory version with uniform syntax must sacrifice the diamond property for abstract reduction. Nevertheless, the analysis does scale to these variants, as we now see.

9.1. 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 the Combined Confluence Theorem easily adapts to this new situation. Figure 7 shows the additions to the syntax of STLC. The proof of **Termination of Abstract Reduction** (Theorem 4.5) 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. To apply the Combined Confluence Theorem, there is still no critical overlap between a-steps, and no possibility of abstract reduction's 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:

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

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

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

Figure 7: Extending STLC with a fixed-point combinator

This can be completed as follows:

L. $E_c[(T \Rightarrow T)\ [T/f]m] \to_a^* E_c[(T \Rightarrow T)\ T] \to_a E_c[T]$
R. $E_c[[\textit{fix } f : T.\, m/f]m] \to_a^* E_c[[((T \Rightarrow T)\ [T/f]m)/f]m] \to_a^* E_c[[T/f]m] \to_a^* E_c[T]$

Here, several steps use $[T/f]m \to_a^* T$, which holds because the term at the peak is typable, using similar reasoning as for Figure 5 in the STLC proof above. This diagram again satisfies the requirement of the third condition of the Combined Confluence Theorem. We conclude Confluence (Theorem 5.2) as above. There are no possible additional stuck terms, so **Progress** (Theorem 8.6) is trivially extended, allowing us to conclude **Type Safety** (Theorem 8.9).

9.2. **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 9.1 (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 $|t'| = t$.*

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

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

Figure 8: Definition of annotation $\hat{\sigma} \star t$

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

$$t_1 \to_a t_2 \to_a \ldots \to_a t_n \to_a T$$

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 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.

$$t_1 \overset{\sigma^1}{\to}_a t_2 \overset{\sigma^2}{\to}_a \ldots \to_a t_n \overset{\sigma^n}{\to}_a T$$

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

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

Given substitution $\hat{\sigma}$ and a term t we define a *annotation* mapping $\hat{\sigma} \star t$ in Figure 8 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:

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

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

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

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

Case: $a(\lambda)$ step. $M(E_a[\lambda x.m] \to_a E_a[\alpha_i \Rightarrow [\alpha_i/x]m])$ unfolds to $(\hat{\sigma} \star E_a[\lambda x.m]) \to_a (\hat{\sigma} \star E_a[\alpha_i \Rightarrow [\alpha_i/x]m])$. Unfolding the definition of $\star$ twice yields $(\hat{\sigma} \star E_a)[\lambda x : \hat{\sigma}(x).\hat{\sigma} \star m] \to_a (\hat{\sigma} \star E_a)[\hat{\sigma}(\alpha_i) \Rightarrow \hat{\sigma} \star ([\alpha_i/x]m)]$. 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 $(\hat{\sigma} \star E_a)[\lambda x : \hat{\sigma}(\alpha_i).\hat{\sigma} \star m] \to_a (\hat{\sigma} \star E_a)[\hat{\sigma}(\alpha_i) \Rightarrow \hat{\sigma} \star ([\alpha_i/x]m)]$. Because $\hat{\sigma}$ contains $x \mapsto \alpha_i$, we can distribute the annotation operation over subsitution $[\alpha_i/x]m$,

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

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

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

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

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

Figure 9: Type inference syntax and semantics

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 $a(\beta)$, $\hat{\sigma}$ 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 the following lemma (whose straightforward proof we omit), the erasure of an annotated term yields the original term.

Lemma 9.2.

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

End Proof.

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

Figure 10: Crucial local diagram for STLC-inf

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

$$(\lambda g.\lambda x.\lambda y.\lambda h.h\ (g\ x)\ (g\ y))\ (\lambda w.w)\ f\ a \quad \to_a^+ \quad \alpha_h(\alpha_g(A \Rightarrow A))(\alpha_g A)$$

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:

$$\alpha_h\ ((A \Rightarrow \alpha_j)\ (A \Rightarrow A))\ \alpha_j \qquad \alpha_h\ \alpha_j\ (((A \Rightarrow A)\ \Rightarrow \alpha_j)\ A)$$

But we are proving **Confluence** only for typeable terms $t \to_{ca}^* T$. 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**.

9.3. **Simply Typed Combinators with Uniform Syntax.** In this section, we consider a language, which we call Uniform-STC, that does not distinguish terms and types syntactically. Advanced type systems like Pure Type Systems must often rely solely on the typing rules to distinguish terms and types (and kinds, superkinds, etc.) [4]. In Uniform-STC, we explore issues that arise in applying the rewriting approach to more advanced type systems. We must now 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

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

Figure 11: Uniform-STLC language syntax and evaluation contexts

$$\begin{array}{ll} c(\beta\text{-}S). & \dfrac{}{E_c[S\langle t_1, t_2, t_3\rangle\ u\ u'\ u''] \to_c E_c[u\ u''\ (u'\ u'')]} \\[2ex] c(\beta\text{-}K). & \dfrac{}{E_c[K\langle t_1, t_2\rangle\ u\ u'] \to_c E_c[u]} \\[2ex] a(S). & S\langle t_1, t_2, t_3\rangle \to_a \mathit{kind}(t_1, \mathit{kind}(t_2, \mathit{kind}(t_3, (t_1 \Rightarrow t_2 \Rightarrow t_3) \Rightarrow (t_1 \Rightarrow t_2) \Rightarrow (t_1 \Rightarrow t_3)))) \\ a(K). & K\langle t_1, t_2\rangle \to_a \mathit{kind}(t_1, \mathit{kind}(t_2, (t_1 \Rightarrow t_2 \Rightarrow t_1))) \\ a(\beta). & (t_1 \Rightarrow t_2)\ t_1 \to_a \mathit{kind}(t_1, t_2) \\ a(k\text{-}\Rightarrow). & \mathit{kind}((t_1 \Rightarrow t_2), t) \to_a \mathit{kind}(t_1, \mathit{kind}(t_2, t)) \\ a(k\text{-}A). & \mathit{kind}(A, t) \to_a t \end{array}$$

Figure 12: Concrete and abstract reduction rules

be described by a first-order term-rewriting system. 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, and indeed, we find it useful to apply the automated confluence prover ACP for testing confluence of our abstract reduction relation [2].

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. As noted above, we are using first-order term-rewriting rules (unlike for previous systems) for abstract reduction.

For STLC (Section 4), 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 $K\langle t_1, t_2\rangle$ 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 can be completed in a way satisfying the requirements of the Combined Confluence Theorem. 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.

Abstract reduction for Uniform-STC does not have the diamond property, due to the non-left-linear rule $a(\beta)$, where there could indeed be redexes in the expressions matching the repeated variable t_1. This could make establishing confluence by hand more difficult, but happily, ACP is able to report that the abstract reduction relation is confluent. So we have dispatched the first condition of the Combined Confluence Theorem automatically.

We then have the following local 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 $\mathit{kind}(t_1, \mathit{kind}(t_2, *))$:

$$\begin{array}{ll} P. & E_c[u[(\hat{t}\ t\ t')]] \leftarrow_a E_c[(K\langle t_1, t_2\rangle\ t\ t')] \rightarrow_c E_c[t] \\ L. & E_c[u[(\hat{t}\ t\ t')]] \rightarrow_a^* E_c[u[(\hat{t}\ t_1\ t'')]] \rightarrow_a E_c[u[((t_2 \Rightarrow t_1)\ t'')]] \rightarrow_a^* \\ & E_c[u[((t_2 \Rightarrow t_1)\ t_2)]] \rightarrow_a E_c[\mathit{kind}(t_1, \mathit{kind}(t_2, t_1))] \rightarrow_a^* E_c[t_1] \\ R. & E_c[t] \rightarrow_a^* E_c[t_1] \end{array}$$

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 can again be seen to be of the form required by the third condition of the Combined Confluence Theorem. 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 9.3 (Type Preservation). *If t has type t_1 and $t \rightarrow_c t'$, then t' also has type 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))
)
```

10. A Rewriting Approach to Normalization for STLC

In this Section, we will prove that every typable (closed) standard term of simply typed lambda calculus has a c-normal form. Our proof of this standard theorem is novel, as it is based on the rewriting approach to typing. We will work with a slightly different presentation of STLC than we saw in Section 4. The syntax we are using in this section is:

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

The abstract and concrete reduction relations are then defined as follows, where we use mixed terms m as contexts (sometimes using meta-variable $\hat{m}$ in this case), writing $m[m']$ to denote the replacement of the unique occurrence of a special variable $*$ in m by m'.

$$\frac{}{\hat{m}[(\lambda x : T.\, m)\ m']\ \to_c\ \hat{m}[[m'/x]m]}\ c(\beta)$$

$$\frac{}{\hat{m}[(T \Rightarrow m)\ T]\ \to_a\ \hat{m}[m]}\ a(\beta)$$

$$\frac{}{\hat{m}[\lambda x : T.\, m]\ \to_a\ \hat{m}[T \Rightarrow [T/x]m]}\ a(\lambda)$$

10.1. **Interpretation of Mixed Terms.** The proof in this section draws upon ideas from standard proofs, such as Girard's proof in the book *Proofs and Types* [8], but develops them in a new way, based on the rewriting approach to typing. As in Girard's proof, we are going to define an interpretation of types as sets of (standard) terms. But here, we will generalize this to interpret mixed terms, as follows. Let WN be the set of standard terms which are weakly normalizing with respect to $\to_c$ (that is, terms t such that there exists some t' such that $t \to_c^* t' \not\to_c$). Also, if $\to$ is any binary relation on standard terms and R any set of standard terms, we will write $\to (R)$ for the image of R under $\to$ (that is, $\{t' \mid \exists t \in R.\ t \to t'\}$).

We first define $\mathcal{R}$ to be the set of all sets R of standard terms satisfying the following conditions:

(1) $R\ = \leftrightarrow_c^* (R)$
(2) $R \neq \emptyset$
(3) $R \subseteq WN$

We will call elements of $\mathcal{R}$ *reducibility candidates*, since the definition is drawn directly from Girard's proof using reducibility candidates. Here, we will not need to apply Girard's technique, but keep the standard terminology. Note that the properties Girard uses in his proof of strong normalization for System F are different from these. In particular, he uses a notion of *neutral terms*, and imposes something equivalent to the more informative statement that the set of neutral terms is contained in R, for the second property. The set of properties we adopt here turns out to be simpler to work with, and are sufficient for normalization of closed terms.

We will use ϕ as a metavariable for functions from *Var* to $\mathcal{R}$, and write $\phi[R/x]$ to mean the function ϕ updated to map variable x to $R \in \mathcal{R}$. Now for any m and ϕ with $FV(m) \subseteq dom(\phi)$, we define the interpretation $[\![m]\!]_\phi$ of m with respect to ϕ in Figure 13. The first two clauses specialize to the usual definition for the reducibility interpretation, when interpreting simple types (where the range type of $T \to m$ is in fact a type T'). The other clauses extend the interpretation from types to mixed terms. To ensure that

$$\begin{array}{lcl}
[\![T \Rightarrow m]\!]_\phi & = & \{t \mid \forall t' \in [\![T]\!]_\phi.\ t\ t' \in [\![m]\!]_\phi\} \\
[\![A]\!]_\phi & = & \mathrm{WN} \\
[\![x]\!]_\phi & = & \phi(x) \\
[\![\lambda x : T.m]\!] & = & \leftrightarrow_c^* (\{\lambda x : T.t \mid \forall t' \in [\![T]\!].\ [t'/x]t \in [\![m]\!]_{\phi[[\![T]\!]/x]}\}) \\
[\![m_1\ m_2]\!]_\phi & = & \leftrightarrow_c^* (\{t_1\ t_2 \mid t_1 \in [\![m_1]\!]_\phi\ \wedge\ t_2 \in [\![m_2]\!]_\phi\}
\end{array}$$

Figure 13: The interpretation of mixed terms

interpretations of types satisfy the first property above of reducibility candidates, we need to close under $\leftrightarrow_c^*$ in the last two clauses of the definition (in Figure 13).

10.2. **Interpretations of Types are Reducibility Candidates.** In this section, we prove that for all types T and ϕ with $FV(T) \subseteq dom(\phi)$, we have $[\![T]\!]_\phi \in \mathcal{R}$. We will elide this condition relating T (or instead m) and ϕ below. We prove the three properties of reducibility candidates given in the previous section. The properties must be proved in order, as later properties depend on earlier ones. The first property is needed in a more general form, for any mixed term m, and not just types T. The second two properties are only needed for types. The proofs in this section are similar to those used for the standard definition of typing, except that there, they are usually proved by mutual induction. Here we can prove them independently, though in sequence, due to the simpler form of the second property. While the development in this section is similar to the usual one, in the next section we will see something significantly different.

Lemma 10.1. $[\![m]\!]_\phi\ = \leftrightarrow_c^* ([\![m]\!]_\phi)$

The proof is by induction on m. If m is a λ-abstraction, or application, the desired property follows by idempotence of $\leftrightarrow_c^*$ as an operator on sets of terms. If m is a variable x, then the property follows by the same property for $\phi(x)$, which we have by assumption. If $\phi = A$, then we must prove

$$WN\ = \leftrightarrow_c^* (\mathrm{WN})$$

Certainly if t is weakly normalizing, and $t' \rightarrow_c^* t$, then t' is also weakly normalizing. If, on the other hand, we have $t \rightarrow_c^* t'$ and $t \in \mathrm{WN}$, then $t' \in \mathrm{WN}$ follows by confluence of $\rightarrow_c$.

Finally, suppose m is $T \Rightarrow m'$ for some m'. Assume an arbitrary $t \in [\![T \Rightarrow m']\!]_\phi$, and arbitrary t' with $t' \leftrightarrow_c^* t$. We must show $t' \in [\![T \Rightarrow m']\!]_\phi$. To do this, by the definition of the interpretation of $\Rightarrow$-terms, it suffices to consider arbitrary $t'' \in [\![T]\!]_\phi$, and show $t'\ t'' \in [\![m]\!]_\phi$. We have $t\ t'' \in [\![m]\!]_\phi$ by the definition of the interpretation of $\Rightarrow$-terms. Then we get the desired conclusion by the induction hypothesis, since $t\ t'' \leftrightarrow^* ct'\ t''$.

Lemma 10.2. $[\![T]\!]_\phi \neq \emptyset$

The proof is by induction on T. If T is A, then the desired property holds immediately, since a variable x, for example, is clearly included in WN. So suppose $T \equiv T_1 \Rightarrow T_2$, for some T_1 and T_2. We must exhibit some $t \in [\![T_1 \Rightarrow T_2]\!]_\phi$. By the induction hypothesis applied to T_2, there exists some $t' \in [\![T_2]\!]_\phi$. Now take $\lambda x : T_1.t'$ for the required term t. We just have to confirm that $\lambda x : T_1.t' \in [\![T_1 \Rightarrow T_2]\!]_\phi$. So assume arbitrary $t'' \in [\![T_1]\!]_\phi$, and show $(\lambda x : T_1.t')\ t'' \in [\![T_2]\!]_\phi$. By Lemma 10.1, it suffices to prove $t' \in [\![T_2]\!]_\phi$, since $(\lambda x : T_1.t')\ t'' \leftrightarrow_c^* t'$. But we are assuming $t' \in [\![T_2]\!]_\phi$.

Lemma 10.3. $[\![T]\!]_\phi \subseteq \text{WN}$

The proof is again by induction on T, and is trivial when T is A. So suppose $T \equiv T_1 \Rightarrow T_2$, and assume arbitrary $t \in [\![T_1 \Rightarrow T_2]\!]_\phi$. We must show $t \in \text{WN}$. By Lemma 10.2, we know there exists some term $t' \in [\![T_1]\!]_\phi$. Then by the definition of the interpretation of $\Rightarrow$-terms, $t\ t' \in [\![T_2]\!]_\phi$. By the induction hypothesis applied to T_2, we then have $t\ t' \in \text{WN}$. But this implies $t \in \text{WN}$, as required.

Corollary 10.4. $[\![T]\!]_\phi \in \mathcal{R}$

The above lemmas have proved that $[\![T]\!]_\phi$ satisfies the three properties for membership in $\mathcal{R}$.

10.3. **The Abstraction Theorem.** In this section, we prove a remarkable theorem, from which the normalization property for typable terms will follow as a corollary. For any mixed terms m and m', and any ϕ with $FV(m) \subseteq dom(\phi)$, we have:

Theorem 10.5 (Abstraction Theorem)**.** $m \to_a m' \implies [\![m]\!]_\phi \subseteq [\![m']\!]_\phi$

Note that well-definedness of $[\![m']\!]_\phi$ in the statement of the theorem follows from the assumption about ϕ and the observation that abstract reduction cannot introduce new variables.

This theorem is remarkable because it reflects the essence of abstraction: the gathering of different concrete entities under the same abstract one. The Abstraction Theorem shows that abstract reduction is increasing the set of concrete terms which are collected under a mixed (and so partially abstract) term. In the next section, we will see how to conclude normalization from this theorem.

We now prove the theorem. By inversion on the assumed derivation of $m \to_a m'$, we see that this judgment must be of the following form, for some (possibly different) m and m':

$$\hat{m}[m] \to_a \hat{m}[m]$$

We begin by induction on $\hat{m}$. Suppose $\hat{m} \equiv m_1\ m_2$, where the hole is in m_1. The case where the hole is in m_2 is similar, so we omit it. To show the required $[\![m_1[m]\ m_2]\!]_\phi \subseteq [\![m_1[m']\ m_2]\!]_\phi$, consider arbitrary $t \in [\![m_1[m]\ m_2]\!]_\phi$. By the definition of the interpretation of applications, we must have $t_1 \in [\![m_1[m]]\!]_\phi$ and $t_2 \in [\![m_2]\!]_\phi$ with $t \leftrightarrow_c^* t_1\ t_2$. Now by the induction hypothesis applied to m_1 we have:

$$[\![m_1[m]]\!]_\phi \subseteq [\![m_1[m']]\!]_\phi$$

This implies $t_1\ t_2 \in [\![m_1[m']\ m_2]\!]_\phi$. From this, we obtain the desired $t \in [\![m_1[m']\ m_2]\!]_\phi$ by the definition of the interpretation of applications.

Suppose now that $\hat{m} \equiv \lambda x : T.m_1$ for some x, T, and m_1. Consider an arbitrary $t \in [\![\lambda x : T.m_1[m]]\!]_\phi$. By the definition of the interpretation of λ-abstractions, this implies that there exists a term t_1 such that $t \leftrightarrow_c^* \lambda x : T.t_1$ and for all $t'' \in [\![T]\!]_\phi$, we have $[t''/x]t_1 \in [\![m_1[m]]\!]_{\phi[[\![T]\!]/x]}$. We must show $t \in [\![\lambda x : T.m_1[m']]\!]_\phi$. By the definition of the interpretation of $\Rightarrow$-terms and Lemma 10.1, it suffices to prove $(\lambda x : T.t_1)\ t'' \in [\![m_1[m']]\!]_{\phi[[\![T]\!]/x]}$ for arbitrary $t'' \in [\![T]\!]_\phi$. Again applying Lemma 10.1, we can see it suffices to prove $[t''/x]t_1 \in [\![m_1[m']]\!]_{\phi[[\![T]\!]/x]}$. This now follows by the induction hypothesis applied to context m_1.

Finally, suppose that the context is $*$, the trivial context. Now we must distinguish the two cases for an abstract reduction.

Case 1. Suppose that we have

$$\lambda x : T.m \ \rightarrow_a \ T \Rightarrow [T/x]m$$

We must prove $[\![\lambda x : T.m]\!]_\phi \subseteq \ [\![T \Rightarrow [T/x]m]\!]_\phi$. So assume arbitrary $t \in [\![\lambda x : T.m]\!]_\phi$, and show $t \in [\![T \Rightarrow [T/x]m]\!]_\phi$. To show that, it suffices to consider arbitrary $t'' \in [\![T]\!]_\phi$, and prove $t\ t'' \in [\![[T/x]m]\!]_\phi$. By the definition of the interpretation of λ-abstractions, we have $t \leftrightarrow_c^* \lambda x : T.t'$, for some t', with $[t''/x]t' \in [\![m]\!]_{\phi[[\![T]\!]/x]}$ for all $t'' \in [\![T]\!]_\phi$. Since $t\ t'' \leftrightarrow_c^* [t''/x]t'$, it suffices by Lemma 10.1 just to prove $[t''/x]t' \in [\![[T/x]m]\!]_\phi$. This follows from the fact just derived, applying also the following lemma:

Lemma 10.6 (Semantic Substitution)**.** $[\![[T/x]m]\!]_\phi \ = \ [\![m]\!]_{\phi[[\![T]\!]/x]}$

The proof of Lemma 10.6 is relegated to the Appendix below.

Case 2. Suppose that we have

$$(T \Rightarrow m)\ T \ \rightarrow_a \ m$$

Assume an arbitrary $t \in [\![(T \Rightarrow m)\ T]\!]_\phi$. By the definition of the interpretation of applications, we then have that there exists $t_1 \in [\![T \Rightarrow m]\!]_\phi$ and $t_2 \in [\![T]\!]_\phi$ such that $t \leftrightarrow_c^* t_1\ t_2$. We must show $t \in [\![m]\!]_\phi$. By the definition of the interpretation of $\Rightarrow$-terms, we obtain $t_1\ t_2 \in [\![m]\!]_\phi$. By Lemma 10.1, this suffices to prove establish $t \in [\![m]\!]_\phi$, since $t \leftrightarrow_c^* t_1\ t_2$.

10.4. Concluding Normalization.

Using the Abstraction Theorem, we can obtain the main result that typable terms are normalizing. First, we need this helper lemma stating that standard terms are in their own interpretations:

Lemma 10.7. *Consider arbitrary t and ϕ, as well as function σ from variables to standard terms. Suppose also that for all $x \in FV$, we have $\sigma(x) \in \phi(x)$. Then we have $\sigma t \in [\![t]\!]_\phi$.*

The proof is by induction on the form of t. If t is a variable x, then we have $\sigma x \in \phi(x)$ by assumption. If t is of the form $\lambda x : T.t_1$, then the definition of the interpretation of mixed terms tells us:

$$[\![\lambda x : T.t_1]\!]_\phi = \leftrightarrow_c^* (\{\lambda x : T.t' \mid \forall t'' \in [\![T]\!]_\phi.\ [t''/x]t' \in [\![t_1]\!]_\phi\})$$

To show that $\lambda x : T.t_1$ is itself a member of the set on the right-hand side of this equation, it suffices to consider an arbitrary $t'' \in [\![T]\!]_\phi$, and show $[t''/x](\sigma t_1) \in [\![t_1]\!]_{\phi[[\![T]\!]_\phi/x]}$. Here we can apply the induction hypothesis for t_1, with $\sigma[t''/x]$ and $\phi[[\![T]\!]_\phi/x]$. The two substitutions still satisfy the required properties. Finally, if t is of the form $t_1\ t_2$, the result easily follows from the induction hypothesis applied to t_1 and also to t_2, and the definition of the interpretation of applications.

Theorem 10.8 (Normalization for Typable Terms)**.** *For all closed standard terms t and types T, if $t \rightarrow_a^* T$, then $t \in$ WN.*

The proof is as follows. By Lemma 10.7, we have $t \in [\![t]\!]_\emptyset$. Then by iterated application of Theorem 10.5, we know that $[\![t]\!]_\emptyset \subseteq [\![T]\!]_\emptyset$. By Lemma 10.3, $[\![T]\!]_\emptyset \subseteq$ WN. Putting these facts together, we get this chain of relationships, which suffices:

$$t \in \ [\![t]\!]_\emptyset \ \subseteq \ [\![T]\!]_\emptyset \ \subseteq \ \text{WN}$$

10.5. **Summary of The Standard Proof.** The standard proof (or one variant of it), which is based on the usual judgment $\Gamma \vdash t : T$ for STLC, proceeds as follows. Here, we summarize Girard's proof of strong normalization, for purposes of comparison. One first defines an interpretation of types:

$$\begin{array}{lcl} t \in \mathit{Red}_b & \Leftrightarrow & t \in \mathit{SN} \\ t \in \mathit{Red}_{T \rightarrow T'} & \Leftrightarrow & \forall t' \in \mathit{Red}_T .\ (t\ t') \in \mathit{Red}_{T'} \end{array}$$

This does not require use of a function ϕ as above (though the standard proof for System F does). For this interpretation of types, one then proves these three properties, by mutual induction on the structure of the type T mentioned in all three properties:

(1) $\mathit{Red}_T(t) \ \Rightarrow\ \mathit{SN}(t)$.
(2) $\mathit{Red}_T(t) \ \Rightarrow\ \mathit{Red}_T(\mathit{next}(t))$.
(3) If t is neutral, then $\mathit{Red}_T(\mathit{next}(t)) \ \Rightarrow\ \mathit{Red}_T(t)$.

A term is neutral iff it is not a λ-abstraction. The third property implies that all the variables are in Red_T for every T. Finally, one derives the following different theorem in place of the Abstraction Theorem:

Theorem 10.9 (Reducibility)**.** *Suppose* $\{x_1 : T_1, \ldots, x_n : T_n\} \vdash t : T$*, and consider arbitrary* $t_i \in \mathit{Red}_{T_i}$*, for all* $i \in \{1, \ldots, n\}$*. Let* σ *be the substitution* $\{(x_1, t_1), \ldots, (x_n, t_n)\}$*. Then* $\sigma\, t \in \mathit{Red}_T$*.*

Now we can obtain as a corollary that $\Gamma \vdash t : T$ implies $t \in \mathit{SN}$, since $\mathit{Red}_b \subseteq \mathit{SN}$ by the first property above, and a substitution σ replacing x by x satisfies the required condition, since all variables are included in all sets Red_T.

10.6. **Discussion.** The main difference in the rewriting-based development and the standard one is in deriving the Abstraction Theorem. The form of the theorem is completely different from Theorem 10.9. One nice technical feature is that for the proof of the Abstraction Theorem, we did not need to apply a substitution to terms inhabiting interpretations of types, as we did for Theorem 10.9. We still needed to use the idea of such a substitution, but it appeared only in a simple helper lemma, namely Lemma 10.7. This is an advantage of the rewriting-based version, since the substitution does not clutter up the proof of the central result. One disadvantage of the rewriting-based version is that we needed the function ϕ and Lemma 10.6 – but this is not such a significant disadvantage, since those devices are needed when we move to System F in the standard development anyway.

11. Conclusion

We have seen how rewriting techniques, in particular confluence analysis, can be used to develop the metatheory of simple types. Typing is treated as a small-step abstract reduction relation, and a generalized form of type safety, based on generalized type preservation and progress theorems, can be established by proving confluence of the combined reduction relation. For this purpose, we have developed a Combined Confluence Theorem, concerning abstract reduction systems, which we have seen can then be used to prove confluence for several different variants of simply typed lambda calculus. We highlight that we are able to cast simple type inference as a form of narrowing, and that for first-order systems, we can apply termination and confluence checkers to automate part of the proof of type preservation. We have also shown how the rewriting approach can be applied to develop a novel proof of normalization for simply typed terms. For this proof, mixed terms are interpreted as sets of standard terms, and a crucial role is played by an Abstract Theorem, showing that those sets are enlarged by reduction of the corresponding interpretations. This proof sheds new light both on the semantics of mixed terms, and the normalization theorem itself.

For future work, the rewriting 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 are currently exploring using local contexts as part of mixed terms, in order to retain the association between x and T locally. We are also quite interested to see how the rewriting approach affects the proof of type preservation for Curry-style System F, as this system has a rather involved type preservation proof, due to the fact that the typing relation is not subject-directed. That is, there are typing rules where the subject (the term being typed) is exactly the same in the premise and conclusion of the rule.

It would be interesting to try to use the rewriting method to automate type preservation proofs completely. 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.

Our longer term goal is to use this approach to design and analyze type systems for symbolic simulation. In program verification tools like PEX and KEY, symbolic simulation is a central component [5, 16]. But these systems do not seek to prove that their symbolic-simulation algorithms are correct. Indeed, the authors of the KEY system argue against expending the effort to do this [6]. 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, and also participants of the RTA 2011 conference for their helpful feedback and suggestions about this work.

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. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 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.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.
[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] Aaron Stump, Garrin Kimmell, and Roba El Haj Omar. Type Preservation as a Confluence Problem. In Manfred Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications (RTA)*, volume 10 of *LIPIcs*, pages 345–360, 2011.
[15] TeReSe, editor. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.
[16] N. Tillmann and W. Schulte. Parameterized Unit Tests. *SIGSOFT Softw. Eng. Notes*, 30:253–262, 2005.
[17] V. van Oostrom. Confluence by decreasing diagrams. *Theoretical Computer Science*, 126(2):259–280, 1994.
[18] A. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.

Appendix A. Proofs of lemmas 4.2, 4.3 and 4.4:

Proof of Lemma 4.2. 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 4.3. The proof is very similar to that of Lemma 4.2, 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 $t_2 \to_a t'_2$ 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:

$$2.\ t_1\ t_2 \to_a t_1\ t'_2$$

Now, we can apply the induction hypothesis on $t'_2 \to^*_a T$, which is shorter than the original reduction sequence in (1), and get:

$$3.\ t_1\ t'_2 \to^*_a t_1\ T$$

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

$$t_1\ t_2 \to^*_a t_1\ T$$

End Proof.

Proof of Lemma 4.4. First we assume $t \to^*_a T'$. We must prove that $T \Rightarrow t \to^*_a T \Rightarrow T'$. The proof is by induction on the number of single steps in the derivation of $t \to^*_a T'$.

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:

$$1. t \to_a t' \to^*_a T'$$

Suppose that the context used for $t \to_a t'$ 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:

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

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

$$3.\ T \Rightarrow t' \to^*_a T \Rightarrow T'$$

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

$$T \Rightarrow t \to^*_a T \Rightarrow T'$$

End Proof.

Appendix B. Proof of Lemma 10.6, Semantic Substitution

Recall the statement of the lemma:

$$[\![[T/x]m]\!]_\phi \ = \ [\![m]\!]_{\phi[[\![T]\!]/x]}$$

The proof is by induction on m, case-splitting on its form as follows. The cases are so direct that we omit the tedious details.

Case $m \equiv x$**.** Then the result is immediate.

Case $m \equiv x' \not\equiv x$**.** Both sides of the equation then equal $\phi(y)$.

Case $m \equiv m_1\ m_2$**.** The result follows directly from the induction hypothesis for m_1 and also for m_2.

Case $m \equiv \lambda x : T.m_1$**.** This also follows directly from the induction hypothesis for m_1.

Case $m \equiv A$**.** Both sides of the equation equal WN.

Case $m \equiv T \Rightarrow m_1$**.** This again follows directly by the induction hypothesis for T and also for m_1.